

NI TestStand™

Reference Manual

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

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

Worldwide Offices

Australia 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 358 (0) 9 725 72511, France 01 57 66 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 41309277, Japan 0120-527196, Korea 82 02 3451 3400, Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466, New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 3390150, Portugal 351 210 311 210, Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222, Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

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

Important Information

Warranty

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

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

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

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

Copyright

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

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

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, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your media, or ni.com/patents.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

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

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

Contents

About This Manual

Conventions	xv
-------------------	----

Chapter 1

NI TestStand Architecture

General Test Executive Concepts	1-1
Major Software Components of TestStand	1-2
TestStand Sequence Editor	1-2
TestStand User Interfaces	1-3
Features Comparison of Sequence Editor and User Interfaces	1-4
TestStand User Interface Controls	1-6
TestStand Engine	1-6
Module Adapters	1-7
TestStand Building Blocks	1-7
Properties	1-8
Built-In and Custom Properties	1-8
Variables	1-8
Standard and Custom Data Types	1-9
Expressions	1-10
Steps	1-10
Step Types	1-11
Sequences	1-11
Step Groups	1-12
Sequence Local Variables	1-12
Sequence Parameters	1-13
Built-in Sequence Properties	1-13
Sequence Files	1-13
Process Models	1-14
Specifying Process Model Files	1-14
Main Sequence and Client Sequence File	1-15
Entry Points	1-15
Automatic Result Collection	1-16
Callback Sequences	1-16
Sequence Executions	1-17

Chapter 2

Sequence Files and Workspaces

Sequence Files	2-1
Types of Sequence Files	2-1
Sequence File Callbacks	2-2
Sequence File Globals.....	2-2
Sequence File Type Definitions	2-2
Comparing and Merging Sequence Files	2-2
Sequences	2-3
Step Groups	2-3
Parameters	2-3
Local Variables	2-4
Sequence File Window and Views.....	2-4
Sequence Hierarchy Window	2-5
Workspaces.....	2-5
Source Code Control.....	2-6
System Deployment	2-6

Chapter 3

Executions

Sequence Context	3-2
Using the Sequence Context	3-2
Lifetime of Local Variables, Parameters, and Custom Step Properties	3-3
Sequence Editor Execution Window	3-3
Executing Sequences	3-4
Using Execution Entry Points	3-4
Executing a Sequence Directly	3-4
Interactively Executing Steps.....	3-5
Debugging Executions	3-5
Terminating and Aborting Executions	3-6
Result Collection	3-7
Custom Result Properties.....	3-8
Exceptions	3-10
Standard Result Properties	3-10
Subsequence Results	3-11
Loop Results	3-13
Report Generation	3-13
Engine Callbacks	3-13
Step Execution.....	3-14

Step Status.....	3-16
Failures	3-17
Terminations.....	3-18
Run-Time Errors	3-18

Chapter 4

Built-In Step Types

Using Step Types	4-1
Built-In Step Properties	4-3
Custom Step Properties	4-5
Custom Properties All Step Types Share	4-6
Step Types You Can Use with Any Module Adapter.....	4-7
Pass/Fail Test.....	4-8
Numeric Limit Test	4-9
Multiple Numeric Limit Test.....	4-11
String Value Test.....	4-13
Action	4-15
Step Types That Work with a Specific Module Adapter.....	4-15
Step Types That Do Not Use Module Adapters	4-16
Flow Control.....	4-16
If	4-16
Else	4-16
Else If	4-17
For	4-17
For Each	4-18
While	4-18
Do While.....	4-18
Break	4-19
Continue	4-19
Select.....	4-19
Case	4-19
Goto.....	4-20
End	4-20
Statement	4-20
Label	4-20
Message Popup.....	4-21
Call Executable.....	4-23
Property Loader	4-24
FTP Files	4-24
Additional Results	4-25
Synchronization Step Types	4-25
Database Step Types.....	4-25

IVI Step Types	4-25
LabVIEW Utility Step Types.....	4-25

Chapter 5

Module Adapters

Configuring Adapters	5-1
Source Code Templates.....	5-2
Search Paths.....	5-2
Configuring Search Paths for Deployment	5-3
LabVIEW Adapter.....	5-4
LabWindows/CVI Adapter.....	5-4
C/C++ DLL Adapter	5-4
Using DLLs	5-5
Using ActiveX Controls in LabVIEW DLLs.....	5-5
Using MFC in DLLs	5-5
Loading Subordinate DLLs.....	5-6
Reading Parameter Information	5-7
Debugging DLLs	5-7
Debugging LabVIEW 8.0 and Later Shared Libraries (DLLs)	5-8
Debugging LabVIEW 7.1.1 Shared Libraries (DLLs).....	5-8
.NET Adapter	5-9
Debugging .NET Assemblies.....	5-9
Using the .NET Framework	5-11
Accessing the TestStand API in Visual Studio .NET 2003 and Visual Studio 2005.....	5-12
ActiveX/COM Adapter	5-12
Debugging ActiveX Automation Servers	5-13
Registering and Unregistering ActiveX/COM Servers.....	5-13
Server Compatibility Options for Visual Basic	5-13
HTBasic Adapter	5-15
Debugging HTBasic Subroutines.....	5-15
Sequence Adapter	5-16
Remote Sequence Execution.....	5-17
Setting up TestStand as a Server for Remote Sequence Execution.....	5-19
Setting Windows System Security	5-19

Chapter 6

Database Logging and Report Generation

Database Concepts	6-1
Databases and Tables	6-1
Database Sessions.....	6-2
Microsoft ADO, OLE DB, and ODBC Database Technologies	6-2
Data Links	6-4
Database Logging Implementation	6-5
Using Database Logging.....	6-6
Logging Property in the Sequence Context.....	6-7
TestStand Database Result Tables	6-8
Default TestStand Table Schema	6-8
Creating Default Result Tables with the Database Viewer	6-9
Adding Support for Other Database Management Systems	6-9
On-the-Fly Database Logging	6-11
Using Data Links	6-11
Using the ODBC Administrator	6-12
Example Data Link and Result Table Setup for Microsoft Access.....	6-12
Database Options—Specifying a Data Link and Schema.....	6-12
Database Viewer—Creating Result Tables.....	6-13
Test Report Implementation	6-14
Using Test Reports.....	6-14
Failure Chain in Reports.....	6-15
Batch Reports	6-15
Property Flags that Affect Reports	6-15
On-the-Fly Report Generation.....	6-16
XML Report Schema.....	6-16

Chapter 7

User Management

Privileges	7-1
Accessing Privilege Settings for the Current User	7-2
Accessing Privilege Settings for Any User	7-3
Defining Custom Privileges	7-3

Chapter 8

Customizing and Configuring TestStand

Tools Menu.....	8-2
TestStand Directory Structure	8-2
<TestStand> Directory	8-2
Components Directory.....	8-3
<TestStand Public> Directory.....	8-5
RuntimeServers Directory	8-6
Copying Read-Only Files to Modify	8-6
<TestStand Application Data> Directory	8-7
Creating String Resource Files	8-7
String Resource File Format	8-8
Configuring Sequence Editor and User Interface Startup Options	8-9
Configure Menu.....	8-11

Chapter 9

Creating Custom User Interfaces

Example User Interfaces.....	9-1
TestStand User Interface Controls.....	9-2
Writing an Application with the TestStand UI Controls	9-3
Manager Controls.....	9-3
Application Manager	9-3
SequenceFileView Manager.....	9-4
ExecutionView Manager	9-4
Visible Controls	9-5
Connecting Manager Controls to Visible Controls.....	9-7
View Connections.....	9-7
List Connections	9-8
Command Connections.....	9-9
Information Source Connections	9-10
Specifying and Changing Control Connections.....	9-12
Editor Versus Operator Interface Applications	9-13
Creating Editor Applications	9-13
License Checking	9-13
Using TestStand UI Controls in Different Environments	9-14
LabVIEW	9-14
LabWindows/CVI	9-14
Microsoft Visual Studio	9-15
Visual C++	9-16
Obtaining an Interface Pointer and CWnd for an ActiveX Control.....	9-17
Using GetDlgItem.....	9-17

Handling Events.....	9-17
Events Typical Applications Handle.....	9-18
ExitApplication	9-18
Wait	9-18
ReportError	9-19
DisplaySequenceFile.....	9-19
DisplayExecution	9-19
Startup and Shutdown	9-20
TestStand Utility Functions Library	9-21
Adding Assembly References in Visual Studio	9-23
Menus and Menu Items.....	9-23
Updating Menus	9-24
Localization	9-25
User Interface Application Styles	9-26
Single Window	9-27
Multiple Window.....	9-27
No Visible Window.....	9-29
Command-Line Arguments	9-29
Persistence of Application Settings	9-30
Configuration File Location	9-30
Adding Custom Application Settings.....	9-31
Documenting Custom User Interfaces	9-31
Deploying a User Interface	9-32
Authenticode Signatures for Windows Vista.....	9-32
Application Manifests	9-33

Chapter 10

Customizing Process Models and Callbacks

Modifying Process Model Sequence Files.....	10-1
Normal Sequences	10-2
Callback Sequences	10-2
Entry Point Sequences	10-3
Modifying Callbacks.....	10-4
Engine Callbacks	10-4
Caveats for Using Engine Callbacks.....	10-8
Front-End Callbacks	10-9

Chapter 11

Type Concepts

Storing Types in Files and Memory	11-1
Modifying Types	11-1
Type Versioning	11-2
Resolving Type Conflicts	11-2
Types Window.....	11-4
Type Palette Files.....	11-4
Sequence Files.....	11-5
Station Globals	11-5
User Manager	11-5

Chapter 12

Standard and Custom Data Types

Using Data Types	12-1
Specifying Array Sizes.....	12-2
Dynamic Array Sizing	12-3
Empty Arrays.....	12-4
Modifying Data Types and Values.....	12-4
Object References	12-5
Using Standard Named Data Types	12-5
Error and CommonResults.....	12-6
Path.....	12-6
Expression	12-7
Creating Custom Data Types.....	12-7
Properties Common to All Data Types	12-7
Custom Properties of Data Types.....	12-8

Chapter 13

Custom Step Types

Creating Custom Step Types	13-1
Properties Common to All Step Types.....	13-2
Step Type Properties Dialog Box.....	13-2
General Tab	13-3
Menu Tab.....	13-3
Substeps Tab.....	13-4
Disable Properties Tab.....	13-5
Code Templates Tab.....	13-6
Version Tab	13-8
Custom Properties of Step Types	13-9
Backward Compatibility.....	13-9

Chapter 14

Deploying TestStand Systems

TestStand System Components	14-1
Setting Up the TestStand Deployment Utility	14-1
Identifying Components to Deploy	14-2
Determining If You Need to Create an Installer	14-2
Creating a System Workspace File.....	14-2
Configuring and Building the Deployment	14-3
Building a Deployment	14-3
Collecting Files.....	14-3
Processing VIs	14-4
Processing Sequence Files.....	14-4
Installing National Instruments Components	14-5
Guidelines for Successful Deployment.....	14-5
Common Deployment Scenarios	14-6
Deploying the TestStand Engine	14-6
Distributing Tests from a Workspace.....	14-7
Adding Dynamically Called Files to a Workspace	14-8
Distributing a User Interface	14-10

Chapter 15

Sequence File Translators

Using a Sequence File Translator	15-1
Creating a Translator DLL.....	15-2
Example Sequence File Translators.....	15-2
Versioning Translators and Custom Sequence Files	15-3
Deploying Translators and Custom Sequence Files	15-4

Appendix A

Process Model Architecture

Appendix B

Synchronization Step Types

Appendix C

Database Step Types

Appendix D

IVI Step Types

Appendix E
LabVIEW Utility Step Types

Appendix F
Technical Support and Professional Services

Index

About This Manual

Use this manual to learn about TestStand concepts and features. Refer to the *NI TestStand System and Architecture Overview Card* for information about how to use the entire TestStand documentation set.

Conventions

The following conventions appear in this manual:

»

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



This icon denotes a note, which alerts you to important information.



This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.

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, controls and buttons on the front panel, dialog boxes, sections of dialog boxes, menu names, and palette names.

italic

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

`monospace`

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

`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.

NI TestStand Architecture

National Instruments recommends that you read the *NI TestStand System and Architecture Overview Card* and the *Using TestStand* manual before you read this manual. National Instruments also recommends that you become familiar with the concepts of this chapter before you proceed through this manual.

General Test Executive Concepts

A test executive organizes and executes sequences of reusable code modules you can create in a variety of programming environments.

This manual uses the following concepts applicable to test executives in general:

- **Code module**—A program module, such as a Microsoft Windows dynamic link library (.dll) or LabVIEW VI (.vi), that contains one or more functions that perform a specific test or other action.
- **Step**—An individual element of a test sequence that can call code modules or perform other operations.
- **Sequence**—A series of steps you specify to execute in a particular order. Whether and when a step executes depends on the results of previous steps.
- **Subsequence**—A sequence another sequence calls as a step.
- **Sequence file**—A file that contains the definition of one or more sequences.
- **Sequence editor**—A program that provides a graphical user interface (GUI) for creating, editing, executing, and debugging sequences.
- **User interface**—A program that provides a GUI for executing sequences on a production station. A sequence editor and user interface can be separate applications or different aspects of the same application.
- **Test executive engine**—A module or set of modules that provide an application programming interface (API) for creating, editing, executing, and debugging sequences. A sequence editor or user interface uses the services of a test executive engine.

- **Application Development Environment (ADE)**—A programming environment, such as LabVIEW, LabWindows™/CVI™, or Microsoft Visual Studio, in which you create code modules and user interfaces.
- **Unit Under Test (UUT)**—The device or component to test.

Major Software Components of TestStand

Refer to the *NI TestStand System and Architecture Overview Card* for a visual representation of how TestStand components interact. You can also refer to the *NI TestStand Help* for more information about each component.



Note If you open help files directly from the <TestStand>\Doc\Help directory, National Instruments recommends that you open TSHelp.chm first because this file is a collection of all the TestStand help files and provides a complete table of contents and index.

TestStand Sequence Editor

The TestStand Sequence Editor is a development environment in which you create, edit, execute, and debug sequences and the tests sequences call. Use the sequence editor to access all TestStand features, such as step types and process models. The sequence editor also includes the following debugging tools you are familiar with in ADEs such as LabVIEW, LabWindows/CVI (ANSI), and Visual Studio:

- Setting breakpoints
- Stepping into, out of, or over steps
- Tracing through program executions
- Displaying variables
- Monitoring variables, expressions, and output messages during executions

In the TestStand Sequence Editor, you can start multiple concurrent executions. You can execute multiple instances of the same sequence, or you can execute different sequences at the same time. Each execution instance opens an Execution window. In Trace Mode, the Execution window shows the steps in the currently executing sequence. If the execution suspends, the Execution window shows the next step to execute and provides debugging options.

The TestStand Sequence Editor contains the following advanced editing features:

- Panes you can dock, float, or hide
- Multiple step editing
- Workspace pane to manage sequence files and test source code
- Source code integration
- Type editing
- Undo and redo edits (except for types)
- Graphical sequence call hierarchy display
- Forward and backward navigation among sequences
- Find and replace
- Integrated sequence file differ
- User management

In the TestStand Sequence Editor, you can fully customize the pane and tab layout to optimize development and debugging tasks. You can also interactively customize the menus, toolbars, and keyboard shortcuts. Refer to the *NI TestStand Help* for more information about working with panes in the sequence editor.

Additionally, you can save custom layouts and reset the layout to the default. TestStand does not automatically save the sequence editor layout from a previous session. Click the **Save Current** button on the UI Configuration tab of the Sequence Editor Options dialog box to save the sequence editor layout with a name you specify. Refer to the *NI TestStand Help* for more information about the Sequence Editor Options dialog box.

TestStand User Interfaces

A TestStand User Interface provides a GUI for executing and debugging test sequences on test stations. User interfaces are designed to protect the integrity of test sequences and are intended for use with deployed custom sequence editors or test systems.

TestStand includes separate user interface applications developed in LabVIEW, LabWindows/CVI, Microsoft Visual Basic .NET, C#, and C++ (MFC). Because TestStand also includes the source code for each user interface, you can fully customize the user interfaces. You can create your own user interface using any programming language that can host ActiveX controls or control ActiveX Automation servers.

With the user interfaces in Editor Mode, you can modify sequences and display sequence variables, sequence parameters, step properties, and so on. With the user interfaces in Operator Mode, you can start multiple concurrent executions, set breakpoints, and single-step through sequences.

Refer to the *NI TestStand System and Architecture Overview Card*, the *NI TestStand User Interface Controls Reference Poster*, and Chapter 9, [Creating Custom User Interfaces](#), for more information about user interfaces.

Features Comparison of Sequence Editor and User Interfaces

Table 1-1 shows the feature differences among the TestStand Sequence Editor, the TestStand User Interfaces in Editor Mode, and the TestStand User Interfaces in Operator Mode.

Table 1-1. Features of TestStand Sequence Editor and TestStand User Interfaces

Features	Application		
	TestStand Sequence Editor	User Interface Editor Mode	User Interface Operator Mode
Environment			
Docking, hiding, and floating panes	✓	—	—
Configurable menus and toolbars	✓	—	—
Navigation among sequences	✓	—	—
Sequence Hierarchy window	✓	—	—
User management configuration	✓	—	—
User privileges enforced	✓	✓	✓
Configurable step list	✓	✓	✓
Workspace support	Dockable pane	Modal dialog	Modal dialog
Source code control support	Integrated	Modal dialog	—
Configure report generation	✓	✓	✓
Configure database logging	✓	✓	✓
Configure station options	✓	✓	✓

Table 1-1. Features of TestStand Sequence Editor and TestStand User Interfaces (Continued)

Features	Application		
	TestStand Sequence Editor	User Interface Editor Mode	User Interface Operator Mode
Editing			
Edit sequence files	✓	✓	—
Insertion Palette	✓	✓	—
Edit steps and modules	Dockable panes	Modal dialogs	—
Integration with ADEs	✓	✓	—
Edit variables and station globals	✓	✓	—
Editing			
Edit types	✓	—	—
Edit process models	✓	✓	—
Multiple step editing	✓	—	—
Undo and redo	✓	✓	—
Find and replace	✓	—	—
Integrated file differ	✓	—	—
Running			
Multithreaded execution	✓	✓	✓
Single-step debugging	✓	✓	✓
Conditional breakpoints	✓	✓	✓
Call stack and thread lists	✓	✓	✓
Variables view	✓	✓	✓
Watch view	✓	—	—
Output messages view	✓	—	—

Table 1-1. Features of TestStand Sequence Editor and TestStand User Interfaces (Continued)

Features	Application		
	TestStand Sequence Editor	User Interface Editor Mode	User Interface Operator Mode
Other			
Can include in deployment	✓	✓	✓
Source code available	—	✓	✓
Minimum license required	TestStand Development System License	TestStand Custom Sequence Editor License	TestStand Base Deployment Engine License

TestStand User Interface Controls

The user interfaces use the TestStand User Interface (UI) Controls, a collection of ActiveX controls for creating custom user interfaces and sequence editors in TestStand. These controls simplify common user interface tasks, such as displaying sequences and executions. You can use these controls in any programming environment that can host ActiveX controls.

Refer to the *NI TestStand Help*, the *NI TestStand User Interface Controls Reference Poster*, and Chapter 9, [Creating Custom User Interfaces](#), for more information about the TestStand UI Controls.

TestStand Engine

The TestStand Engine is a set of DLLs that exports an ActiveX Automation API. The TestStand Sequence Editor and User Interface Controls use the TestStand API, which you can call from any programming environment that supports access to ActiveX Automation servers, including code modules you write in LabVIEW and LabWindows/CVI.

Refer to the *NI TestStand Help* for more information about the TestStand API.

Module Adapters

The TestStand Engine uses module adapters to invoke code modules TestStand sequences call. Module adapters load and call code modules, pass parameters to code modules, and return values and status from code modules. TestStand includes the following module adapters to obtain the list of parameters the code module requires:

- **LabVIEW Adapter**—Calls LabVIEW VIs with a variety of connector panes.
- **LabWindows/CVI Adapter**—Calls C functions with a variety of parameter types in source files in the current LabWindows/CVI project, object files, library files, or DLLs.
- **C/C++ DLL Adapter**—Calls functions or methods in a DLL with a variety of parameter types, including National Instruments Measurement Studio classes.
- **.NET Adapter**—Calls methods and accesses the properties of objects in a .NET assembly.
- **ActiveX/COM Adapter**—Calls methods and accesses the properties of objects in an ActiveX server.
- **HTBasic Adapter**—Calls HTBasic subroutines.
- **Sequence Adapter**—Calls other TestStand sequences with parameters.

The module adapters contain other important information in addition to the calling convention and parameter lists. ADE-specific module adapters can open the ADE, create source code for a new code module in the ADE, and display the source for an existing code module in the ADE.

Refer to Chapter 5, *Module Adapters*, for more information about module adapters.

TestStand Building Blocks

You use properties, variables, data types, expressions, steps, sequences, sequence files, process models, result collection, and sequence executions to create test systems. Refer to the *NI TestStand System and Architecture Overview Card* for a visual representation of how these building blocks interact.

Properties

A property is a storage space for information and can store a single value or a multidimensional array of values of the same data type.

A value can be a number, string, Boolean, .NET object reference, or ActiveX object reference. TestStand stores numbers as 64-bit, floating-point values in the IEEE 754 format. Values are not containers and thus cannot contain subproperties.

TestStand uses the following major categories of properties, defined by the kinds of values the properties contain:

- **Single-valued property**—Contains a single value. TestStand supports number, string, Boolean, and object reference single-valued properties.
- **Array property**—Contains an array of values. TestStand supports number, string, Boolean, and object reference array properties.
- **Property-array property**—Contains a value that is an array of subproperties of a single type.
- **Container property**—Contains no values but contains multiple subproperties. Container properties are analogous to clusters in LabVIEW and to structures in C/C++.

Built-In and Custom Properties

TestStand defines a set of built-in properties for some objects, such as steps and sequences. The TestStand Sequence Editor hides these built-in properties by default, but you can modify the property values through panes and dialog boxes. You can also access the built-in properties through the TestStand API.

You can define new custom properties, such as high- and low-limit properties in a step or local variables in a sequence.

Variables

Variables are properties you can freely create in certain contexts. Variables can apply globally to a sequence file or locally to a particular sequence. You can also use station global variables with values that persist across different executions and across different invocations of the sequence editor or user interfaces. The TestStand Engine maintains the value of station global variables in a file on the computer on which you installed the TestStand Engine.

You can use TestStand variables to share data among tests written in different programming languages, even if the data representations are incompatible. You can pass values you store in variables and properties to code modules. You can also use the TestStand API to access variable and property values directly from code modules.

Each step in a sequence can include properties. The type of step determines its set of properties. Refer to the [Step Types](#) section of this chapter for more information about types of steps.

When executing sequences, TestStand maintains a `SequenceContext` object that contains references to all global variables, all local variables, and all step properties in active sequences. The content of the `SequenceContext` object changes according to the currently executing sequence and step. If you pass a `SequenceContext` object reference to a code module, you can use the code module to access information stored within the `SequenceContext` object.

Standard and Custom Data Types

When you create a variable or property, you specify its data type. In some cases, you use a built-in data type, such as a number or a Boolean. In other cases, you might want to use an arbitrarily complex data structure by defining a custom named data type you can reuse with other variables or properties. When you define your own named data type, the data type must use a unique name. You can add or delete subproperties in each named data type you create without restriction. For example, you might create a Transmitter data type that contains subproperties such as `NumChannels` and `PowerLevel`.

TestStand defines a set of standard named data types, which include `Error`, `CommonResults`, `Path`, and `Expression`. You can add subproperties to some standard named data types, but you cannot delete any of the built-in subproperties.



Note Modifying the standard named data types might result in type conflicts when you open other sequence files that reference these types. Refer to Chapter 12, [Standard and Custom Data Types](#), for more information about the standard named data types.

Although each variable or property you create with a named data type has the same data structure, the variable or property can contain different values.

Expressions

You can use the values of variables and properties in numerous ways, such as passing a variable to a code module or using a property value to determine whether to execute a step. For these same purposes, you can use an expression, which is a formula that calculates a new value from the values of multiple variables or properties. In expressions, you can access all variables and properties active in the sequence context when TestStand evaluates the expression.

You can use an expression wherever you would use a simple variable or property value. TestStand supports all applicable expression operators and syntax you can use in C, C++, Java, and Visual Basic .NET. You can also call the TestStand API directly from within expressions.

The following is an example of an expression:

```
Locals.MidBandFrequency = (Step.HighFrequency +  
    Step.LowFrequency) / 2
```



Note Accessing the TestStand API from within expressions is slightly slower than using multiple ActiveX/COM Adapter steps to perform similar operations.

All TestStand controls that accept expressions provide context-sensitive editing features, such as drop-down lists, syntax checking, and expression coloring to help you create expressions.

Refer to the *NI TestStand Help* for more information about TestStand expressions.

Steps

TestStand steps can perform many actions, such as initializing an instrument, performing a complex test, or affecting the flow of execution in a sequence. Steps perform these actions through several types of mechanisms, including jumping to another step, executing an expression, calling a subsequence, or calling an external code module.

Steps can include built-in and custom properties. For steps that call code modules, the TestStand adapter uses the built-in step properties to store parameters to pass to the code module and to specify where to store results the code module returns.

Not all steps call code modules. Some steps perform standard actions you configure using panes and dialog boxes. In this case, the panes and dialog boxes use the custom step properties to store the configuration settings you specify.

Step Types

Just as each property or variable has a data type, each step has a step type. Each step of a type includes the built-in step properties and any number of custom step properties. Although all steps of the same type have the same properties, the values of those properties can differ. The step type specifies the initial values of all the step properties. Refer to Chapter 4, *Built-In Step Types*, for descriptions of the predefined step types.

You can create a test application using only the predefined step types, and you can also create your own custom step types to define standard, reusable classes of steps that apply specifically to the application. Refer to Chapter 13, *Custom Step Types*, for more information about creating your own step types.

Source Code Templates

You can define a source code template for a new step type. When you create a new step of a particular type, you can use a source code template to generate source code for the code module of the step. You can specify different source code templates for different module adapters.

Sequences

A sequence consists of a series of steps. A TestStand sequence can consist of the following components:

- Setup step group
- Main step group
- Cleanup step group
- Sequence local variables
- Parameters
- Built-in sequence properties
- Callback sequences

Step Groups

TestStand executes the steps in the Setup step group first, the Main step group second, and the Cleanup step group last. The Setup step group typically contains steps that initialize instruments, fixtures, or a UUT. The Main step group typically contains the bulk of the steps in a sequence, including the steps that test the UUT. The Cleanup step group typically contains steps that power down or restore the initial state of instruments, fixtures, and the UUT.

Use separate step groups to ensure that the steps in the Cleanup step group execute regardless of whether the sequence completes successfully or a run-time error occurs in the sequence. If a step in the Setup or Main step group generates a run-time error, the flow of execution jumps to the Cleanup step group. The cleanup steps always run even if some of the setup steps do not run. If a cleanup step causes a run-time error, execution continues to the next cleanup step.

If a run-time error occurs in a sequence, TestStand reports the run-time error to the calling sequence. Execution in the calling sequence jumps to the Cleanup step group in the calling sequence. This process continues up the call stack to the top-level sequence. Thus, when a run-time error occurs, TestStand terminates execution after running all the cleanup steps in all the sequences in the sequence call stack.

Sequence Local Variables

You can create an unlimited number of local variables in a sequence. Use local variables to store data relevant to the execution of the sequence. You can pass local variables by value or by reference to any step in the sequence that calls a subsequence or any step that calls a code module that uses the LabVIEW, LabWindows/CVI, C/C++ DLL, .NET, or ActiveX/COM Adapter. You can also access local variables from code modules of steps in the sequence using the TestStand API.



Note TestStand can pass data to LabVIEW VIs only by value. LabVIEW does not support passing data by reference. You can return a value as an indicator, which TestStand treats as a separate parameter.

Sequence Parameters

Each sequence includes its own list of parameters. Use these parameters to pass data to a sequence when you call the sequence as a subsequence. Using parameters in this way is analogous to wiring data to terminals when you call a subVI in LabVIEW and to passing arguments to a function call in LabWindows/CVI. You can also specify a default value for each parameter.

You can specify the number of parameters and the data type of each parameter. You can select a value to pass to the parameter or use the default value of the parameter. You can pass sequence parameters by value or by reference to any step in the sequence that calls a subsequence or any step that calls a code module that uses the LabVIEW, LabWindows/CVI, C/C++ DLL, .NET, or ActiveX/COM Adapter. You can also access parameters from code modules of steps in the sequence by using the TestStand API.



Note TestStand can pass data to LabVIEW VIs only by value. LabVIEW does not support passing data by reference. You can return a value as an indicator, which TestStand treats as a separate parameter.

Creating parameterized code modules can be helpful when you develop a TestStand system because you can reuse the code modules with different parameters in future systems. Carefully analyze use cases and create extensible code modules accordingly. Be aware that code modules you have parameterized too much can be difficult to use and maintain.

Built-in Sequence Properties

Sequences include built-in properties you can specify using the Sequence Properties dialog box. For example, you can specify that the flow of execution jumps to the Cleanup step group whenever a step sets the status of the sequence to `Failed`.

Refer to the *NI TestStand Help* for more information about the Sequence Properties dialog box.

Sequence Files

Sequence files can contain one or more sequences. Sequence files can also contain global variables, which all sequences in the sequence file can access.

Sequence files include built-in properties. Use the Sequence File Properties dialog box to specify values for the built-in properties. For example, you

can specify Load and Unload Options that override the Load and Unload Options of all the steps in all the sequences in the file.

Refer to the *NI TestStand Help* for more information about the Sequence File Properties dialog box.

Process Models

Testing a UUT requires more than just executing a set of tests. Usually, the test system must perform a series of operations before and after it executes the sequence that performs the tests. Common operations that define the testing process include identifying the UUT, notifying the operator of pass/fail status, logging results, and generating a test report. The set of such operations and their flow of execution is called a process model.

Some commercial test executives implement their process model internally and do not allow you to modify the model. Other test executives do not come with a process model at all. TestStand includes a predefined Sequential model, Parallel model, and Batch model you can modify or replace. Use the Sequential model to run a test sequence on one UUT at a time. Use the Parallel and Batch models to run the same test sequence on multiple UUTs at the same time.

With TestStand, you can define your own process model, which is a sequence file in which you can write different test sequences without repeating standard testing operations in each sequence. The ability to modify a process model is essential because the testing process can vary according to production lines, production sites, or company systems and practices. You can edit a process model in the same way you edit other sequence files.

You can use client sequence files to customize various model operations by overriding the callback sequences process models define. Refer to the [Modifying Process Model Sequence Files](#) section of Chapter 10, [Customizing Process Models and Callbacks](#), for more information about customizing model operations.

Specifying Process Model Files

The station model file is the process model file TestStand uses for all sequence files. The Sequential model is the default station model file. Use the Station Options dialog box to select a different station model and to specify if individual sequence files can use their own process model files.

If you allow individual sequence files to specify their own process model files, use the Sequence File Properties dialog box to set the process model file for the sequence file. You can also specify if a sequence file does not use a process model.

Refer to the *NI TestStand Help* for more information about the Station Options dialog box and the Sequence File Properties dialog box.

Main Sequence and Client Sequence File

The Main sequence initiates the tests on a UUT. The process model defines what is constant about the testing process, and Main sequences define the unique steps for the different types of tests to run. When you create a new sequence file, TestStand automatically inserts a Main sequence in the file. You must name each Main sequence `MainSequence`. The process model invokes the Main sequence as part of the overall testing process. TestStand determines which process model file to use with the Main sequence. TestStand uses the station model file unless the sequence file specifies a different process model file and you enabled the Allow Other Models option in the Station Options dialog box to allow sequence files to override the station model setting.

After TestStand identifies the process model to use with the Main sequence, the file that contains the Main sequence becomes a client sequence file of the process model.

Entry Points

A process model defines a set of entry points, and each entry point is a sequence in the process model file. Defining multiple entry points in a process model gives the test station operator different ways to invoke a Main sequence or configure the process model.

The sequence for a process model entry point can contain calls to DLLs, subsequences, Goto steps, and so on. You can specify two types of entry points—Execution entry points and Configuration entry points.

Refer to the [Using Execution Entry Points](#) section of Chapter 3, [Executions](#), for more information about entry points.

Automatic Result Collection

TestStand can automatically collect the results of each step. You can enable or disable result collection for a step, a sequence, an execution, or for the entire test station.

Each sequence includes a local array that stores the results of each step. The content of the results for each step varies depending on the step type. TestStand stores the results for a step in the array and adds information, such as the name of the step and its position in the sequence. For a step that calls a sequence, TestStand also adds the result array from the subsequence.

Refer to the [Result Collection](#) section of Chapter 3, [Executions](#), for more information about how TestStand collects results. Refer to Chapter 6, [Database Logging and Report Generation](#), for more information about report generation and database logging features for processing the collected test results.

Callback Sequences

Callbacks are sequences TestStand calls under specific circumstances. You can create new callback sequences or you can override existing callbacks to customize the operation of the test station. Use the Sequence File Callbacks dialog box to add a callback sequence to a sequence file.

Refer to the *NI TestStand Help* for more information about the Sequence File Callbacks dialog box.

TestStand defines Model callbacks, Engine callbacks, and Front-End callbacks based on the entity that invokes the callback and the location in which you define the callback, as shown in Table 1-2. Use Model callbacks to customize the behavior of a process model for each Main sequence that uses it. The TestStand Engine defines and invokes Engine callbacks at specific points during execution. User interface programs call Front-End callbacks so multiple user interfaces can share the same behavior for a specific operation.

Table 1-2. Callback Types

Callback Type	Where You Define the Callback	What Calls the Callback
Model Callbacks	The process model file defines Model callbacks, and the client sequence file or <code>StationCallbacks.seq</code> can override the callback	Sequences in the process model file
Engine Callbacks	<code>StationCallbacks.seq</code> for Station Engine callbacks, the process model file for Process Model Engine callbacks, or a regular sequence file for Sequence File Engine callbacks	Engine
Front-End Callbacks	<code>FrontEndCallbacks.seq</code>	User interface application

Sequence Executions

When you run a sequence, TestStand creates an `Execution` object that contains all the information TestStand needs to run the sequence and the subsequences it calls. While an execution is active, you can start another execution by running the same sequence again or by running a different one. TestStand does not limit the number of executions you can run concurrently. An `Execution` object initially starts with a single execution thread. You can use sequence call multithreading options to create additional threads within an execution or to launch new executions. An execution groups related threads so that setting a breakpoint suspends all threads in the execution. In the same way, terminating an execution also terminates all threads in the execution.

Sequence Files and Workspaces

Sequence files and workspaces help you organize your work.

Sequence Files

A TestStand sequence file (.seq) contains any number of sequences, a set of types the sequence file uses, and any global variables steps and sequences in the file share.

Types of Sequence Files

TestStand includes the following types of sequence files:

- **Normal**—Contains sequences that test UUTs
- **Model**—Contains process model sequences
- **Station Callback**—Contains Station callback sequences
- **Front-End Callback**—Contains Front-End callback sequences

Most sequence files you create are normal sequence files. Usually, an application has one Station callback sequence file and one Front-End callback sequence file.

Normal sequence files specify if they always use the station process model, a specific process model, or no process model.

From within the TestStand Sequence Editor, use the Sequence File Properties dialog box to set the type of sequence, the sequence file process model settings, and other sequence file properties.

Refer to the *NI TestStand Help* for more information about the Sequence File Properties dialog box.

Sequence File Callbacks

TestStand uses callback sequences under specific circumstances. Sequence files can contain sequences that override these callback sequences. Use the Sequence File Callbacks dialog box to specify these callback sequences.

Refer to the *NI TestStand Help* for more information about the Sequence File Callbacks dialog box. Refer to Chapter 10, [Customizing Process Models and Callbacks](#), for more information about callbacks and overriding callback sequences.

Sequence File Globals

Each sequence file can contain any number of global variables, which you can access from any step or sequence within the sequence file in which you define the global variables. View and edit the global variables in the Variables pane. Use the Value column in the Variables pane to modify string, numeric, and Boolean values.

Refer to the *NI TestStand Help* for more information about the Variables pane.

Sequence File Type Definitions

Sequence files contain the type definitions for every step, property, and variable the sequence file contains. View and edit the types a sequence file contains in the Types pane.

Refer to the *NI TestStand Help* for more information about the Types pane. Refer to Chapter 11, [Type Concepts](#), for more information about types and type editing.

Comparing and Merging Sequence Files

The Sequence File Differ is a stand-alone application and a tool within the sequence editor you can use to compare and merge differences between two sequence files. The Sequence File Differ compares the sequence files and presents the differences in a separate, two-pane window.

Refer to the *NI TestStand Help* for more information about the Differ window and Sequence File Differ application.

Sequences

Each sequence can contain steps, parameters, and local variables. View and edit the list of sequences in the Sequences pane of the Sequence File window. View and edit the contents of a selected sequence in the Steps pane of the Sequence File window.

Sequences have properties you can view and edit in the Sequence Properties dialog box. Refer to the *NI TestStand Help* for more information about the Sequence Properties dialog box.

Step Groups

Sequences contain steps in a Setup group, a Main group, and a Cleanup group. You can view and edit the step groups in the Steps pane of the Sequence File window.

Use the Setup step group for steps that initialize or configure instruments, fixtures, and UUTs. Use the Main step group for steps that test the UUTs. Use the Cleanup step group for steps that power down or release handles to instruments, fixtures, and UUTs.

Refer to the *NI TestStand Help* for more information about the Steps pane.

Parameters

Each sequence has its own list of parameters. Use these parameters to pass data to and from a sequence when you call the sequence as a subsequence. View and edit the parameters for a sequence in the Variables pane of the Sequence File window. Use the Value column in the Variables pane to modify string, numeric, and Boolean values.

Refer to the *NI TestStand Help* for more information about the Variables pane.

Local Variables

Use local variables to store data relevant to the execution of the sequence. You can access local variables from within steps and code modules. You can also use local variables for maintaining counts, holding intermediate values, or any other purpose. View and edit the local variables in the Variables pane. Use the Value column in the Variables pane to modify string, numeric, and Boolean values.

Refer to the *NI TestStand Help* for more information about the Variables pane.

Sequence File Window and Views

Within the TestStand Sequence Editor, you can view and edit sequence files in the Sequence File window, as shown in Figure 2-1.

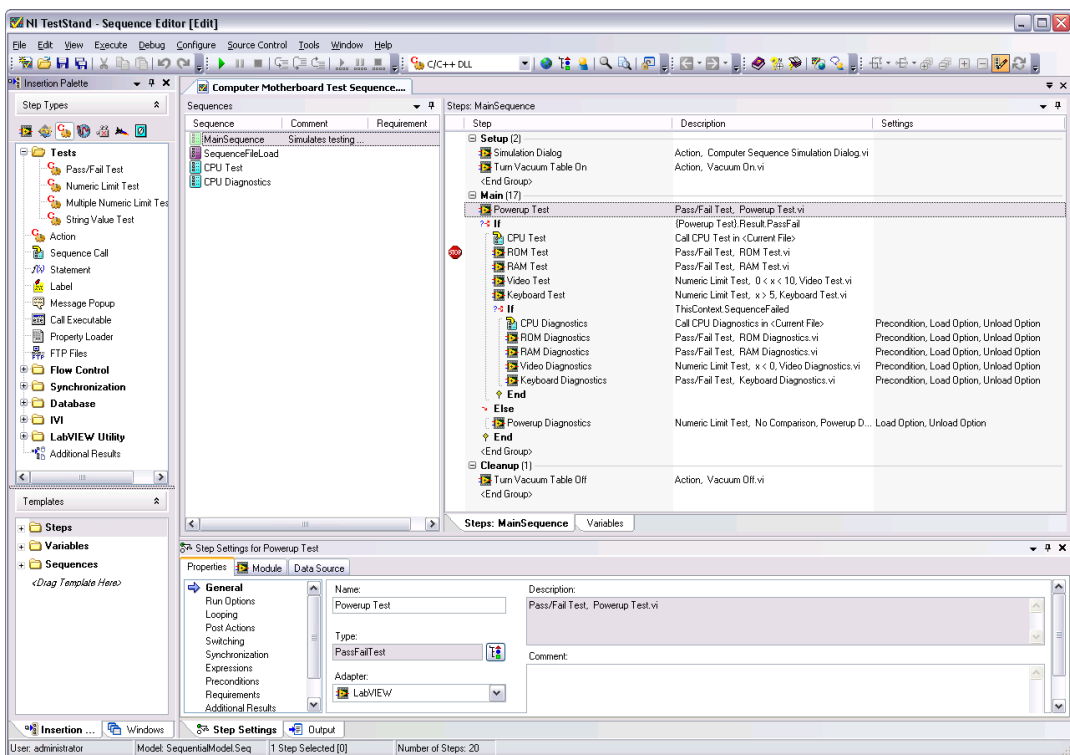


Figure 2-1. Sequence File Window and Insertion Palette

To open an existing sequence file in the Sequence File window, select **File»Open Sequence File**. To create a new Sequence File window, select **File»New Sequence File**.

The Sequence File window contains the following panes:

- **Sequences**—Displays a list of sequences in a file. Use this pane to create new sequences and to cut, copy, and paste sequences.
- **Steps**—Displays the steps in a specific sequence. Expand the Setup, Main, or Cleanup group to view its contents.
- **Variables**—Displays the variables the steps you select in the Steps pane can access at run time. The variables include locals, parameters, file globals, station globals, and run state information accessible to TestStand when the sequence executes.

Refer to the *NI TestStand Help* for more information about the Sequence File window and its panes.

Sequence Hierarchy Window

TestStand sequences can use the Sequence Call step type to call subsequences. The Sequence Hierarchy window shows the relationship between sequences and subsequences by displaying a graph that represents all the sequence calls that start at a root sequence. Each sequence appears as a node in the graph, and each sequence call step appears as a link between nodes.

Refer to the *NI TestStand Help* for more information about accessing the Sequence Hierarchy window and using the buttons on the Sequence Hierarchy toolbar. Refer to Chapter 4, *Built-In Step Types*, for more information about the Sequence Call step type.

Workspaces

Create a workspace to organize and access development files. Use workspaces early in development so you can easily keep track of files while you are developing. A TestStand workspace file (.tsw) contains references to any number of TestStand project files. A TestStand project file (.tpj) contains references to any number of other files of any type.

Use TestStand project files to organize related files in the test system. You can insert any number of files into a project. You can also insert folders in a project to contain files or other folders.

In the sequence editor, use the Workspace pane to view and edit a workspace file and the project files it references. You can open only one workspace file at a time. To open an existing workspace file, select **File»Open Workspace File**. To create a new workspace file, select **File»New Workspace File**.



Note If you modify or replace a file in a workspace, TestStand reflects the changes only if the file maintains the same filename and either the file maintains the same path if the workspace uses an absolute path to locate the file or the workspace can still locate the file using the TestStand search paths. Refer to the [Search Paths](#) section of Chapter 5, [Module Adapters](#), for more information about TestStand search directories.

The TestStand Deployment Utility also uses a workspace to specify the files to include in the deployment image or installer the utility creates.

Refer to the *NI TestStand Help* for more information about the Workspace pane.

Source Code Control

Use workspace files early in development to easily access files in a source code control (SCC) system. To perform SCC operations on files from within TestStand, select an SCC provider on the Source Control tab of the Station Options dialog box and configure the SCC settings for the workspace on the Source Control tab of the Workspace Object Properties dialog box.



Note National Instruments tested TestStand with Microsoft Visual SourceSafe, Perforce, MKS Source Integrity, Rational ClearCase, and Merant PVCS.

Refer to the *NI TestStand Help* for more information about using SCC tools with TestStand.

System Deployment

You must create a workspace to use the TestStand Deployment Utility, which uses workspace and project files to collect all the files required to successfully distribute a test system to a target computer. The deployment utility creates an image or an installer for the test system.

Refer to Chapter 14, [Deploying TestStand Systems](#), for more information about system deployment and the TestStand Deployment Utility.

Executions

An execution is an object that contains all the information TestStand uses to run a sequence and subsequences. When an execution is active, you can start other executions by running the same sequence again or by running different sequences. TestStand does not limit the number of executions you can run concurrently. An execution can start with a single thread and then launch additional threads. When you suspend, terminate, or abort an execution, you stop all threads in the execution.

When TestStand begins executing a sequence, it makes a run-time copy of the sequence local variables and the custom properties of the steps in a sequence. If the sequence calls itself recursively, TestStand creates a separate run-time copy of the local variables and custom step properties for each running instance of the sequence. Modifications to the values of local variables and custom step properties apply only to the run-time copy and do not affect the sequence file in memory or on disk.



Note TestStand shares built-in properties of steps and sequences at run time. For these shared properties, TestStand does not create a unique run-time copy but instead references the edit-time copy. Any changes to the run-time reference of these built-in properties edits the original *Step* or *Sequence* object in the sequence file.

For each execution thread, TestStand maintains an execution pointer that points to the current step, a call stack, and a run-time copy of the local variables and custom properties for all sequences and steps on the call stack.

The Execution tab of the Station Options dialog box provides a number of execution options that control tracing, breakpoints, and result collection. Refer to the *NI TestStand Help* for more information about the Execution tab of the Station Options dialog box.

Sequence Context

Before executing the steps in a sequence, TestStand creates a run-time copy of the sequence, which allows TestStand to maintain separate local variable and step property values for each sequence invocation.

TestStand maintains a sequence context that contains references to the run-time copy of the sequence, to all global variables, and to step properties in the active sequence. The content of a sequence context varies depending on the currently executing step. Refer to the *NI TestStand Help* for more information about the content of the sequence context.

Using the Sequence Context

In expressions, you can access the value of a variable or property by specifying a path from the sequence context to the particular variable or property. For example, you can set the status of a step using the following expression:

```
Step.Result.Status = "Passed"
```

During an execution, you can view and modify the values of the properties in the sequence context from the Variables pane in the Execution window. The Variables pane displays the sequence context for the sequence invocation currently selected in the Call Stack pane. You can also monitor individual variables or properties from the Watch View pane. Refer to the *NI TestStand Help* for more information about using the Variables pane, Watch View pane, and Call Stack pane of the Execution window.

You can pass a reference to a `SequenceContext` object to a code module. In code modules, you access the value of a variable or property by using `PropertyObject` methods in the TestStand API with the sequence context. As with expressions, you must specify a path from the sequence context to the particular property or variable. Refer to the *NI TestStand Help* for more information about accessing the properties in the sequence context from code modules.

Select **View»Sequence File»Variables** or **View»Execution»Variables** in the sequence editor to open the Variables pane, which contains the names of variables, properties, and sequence parameters you can access from expressions and code modules. Refer to the *NI TestStand Help* for more information about the Variables pane.



Note Some properties are not populated until run time.

Lifetime of Local Variables, Parameters, and Custom Step Properties

Multiple instances of a sequence can run at the same time, such as when you call a sequence recursively or when a sequence runs in multiple concurrent threads. For each instance of the sequence, TestStand creates a copy of the sequence parameters, local variables, and custom properties of each step. When a sequence completes, TestStand discards the values of the parameters, local variables, and custom properties.

Sequence Editor Execution Window

The sequence editor displays each execution in a separate Execution window, as shown in Figure 3-1.

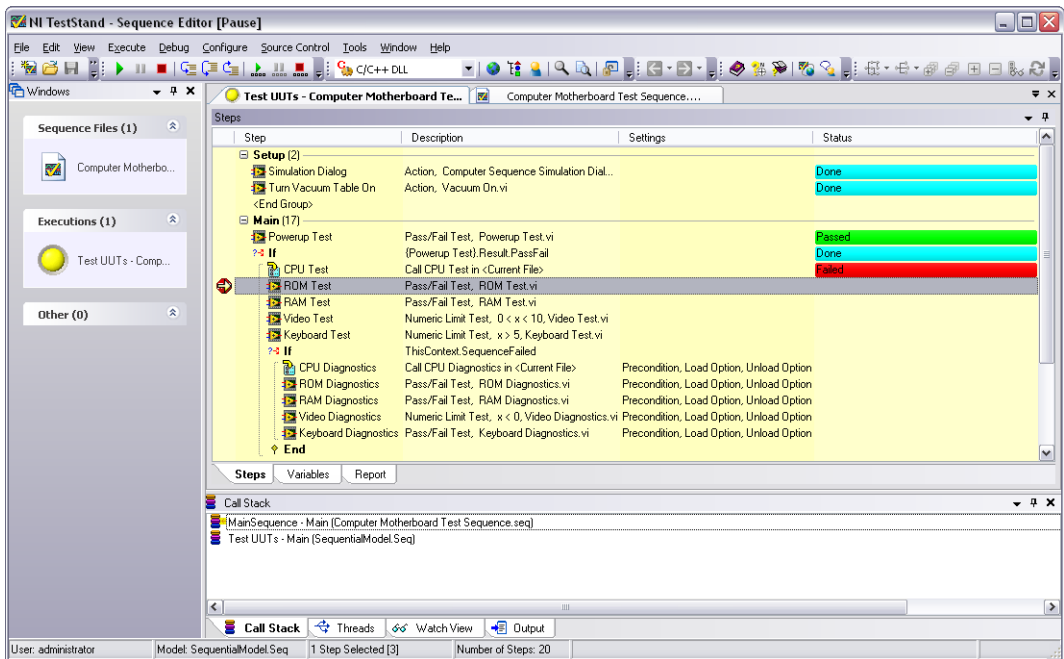


Figure 3-1. Sequence Editor Execution Window

Refer to the *NI TestStand Help* for more information about the Execution window.

Executing Sequences

You can initiate an execution by launching a sequence through an Execution entry point, by launching a sequence directly, or by executing a group of steps interactively. You can debug executions, and you can terminate or abort executions.

Using Execution Entry Points

You can use an Execution entry point to start an execution only when a sequence file that contains a sequence with the name `MainSequence` occupies the active window. The Execute menu of the sequence editor includes a list of Execution entry points.

Each Execution entry point in the menu represents a separate entry point sequence in the process model that applies to the active sequence file. When you select an Execution entry point from the Execute menu, you actually run an entry point sequence in a process model file. The Execution entry point sequence invokes the Main sequence one time or multiple times.

Execution entry points in a process model provide different ways for the test station operator to invoke a Main sequence. Execution entry points handle common operations, such as UUT identification and test report generation. For example, the default TestStand process model provides two Execution entry points—Test UUTs and Single Pass. The Test UUTs Execution entry point initiates a loop that repeatedly identifies and tests UUTs. The Single Pass Execution entry point tests a single UUT without identifying it.

Refer to Chapter 10, *Customizing Process Models and Callbacks*, and Appendix A, *Process Model Architecture*, for more information about using process models.

Executing a Sequence Directly

To execute a sequence without using a process model, select **Execute» Run <Sequence Name>**, where **<Sequence Name>** is the name of the current sequence. Executing a sequence directly skips the process model operations, such as UUT identification and test report generation. You can use this method to execute any sequence. Executing a sequence directly is helpful for performing unit testing or debugging.

Interactively Executing Steps

To interactively execute selected steps in a sequence, select **Run Selected Steps** or **Loop On Selected Steps** from the context menu or from the **Execute** menu in the sequence editor or user interfaces.

You can run steps the following two ways in interactive mode:

- Run steps interactively from a Sequence File window to create a new execution called a root interactive execution. You can set station options to control if the Setup and Cleanup step groups of the sequence run as part of a root interactive execution.
- Run steps interactively from an existing Execution window for a normal execution suspended at a breakpoint. The selected steps run in a nested interactive execution within the context of the normal execution.

The steps you run interactively can access the variable values of the normal execution and add to the results. If you used the process model for the original execution, the test report includes these results. When the selected steps complete, the execution returns to the originally suspended step. A configurable station option specifies if step failures and errors propagate to the original execution.

In interactive mode, the selected steps run in the order in which they appear in the sequence.

To configure TestStand to evaluate preconditions when executing interactively, select **Configure»Station Options** and enable the **Evaluate Preconditions** option in the Interactive Executions section of the Execution tab of the Station Options dialog box. You can also use the Branching Mode control in this dialog box to configure if interactive executions branch to unselected steps. Refer to the *NI TestStand Help* for more information about the Station Options dialog box.

Debugging Executions

Use the following panes to gather information when you single-step through an execution:

- **Threads and Call Stack panes**—The Threads pane displays the threads running in the execution and selects the active thread to view. The Call Stack pane displays the sequence invocations for the active thread and selects the active sequence invocation to view. Refer to the *NI TestStand Help* for more information about the Threads and Call Stack panes.

- **Variables pane**—Displays the sequence context for the sequence invocation currently selected in the Call Stack pane. The sequence context contains all the variables and properties the steps in the selected sequence invocation can access. Use the Variables pane to examine and modify the values of these variables and properties. Refer to the *NI TestStand Help* for more information about the Variables pane.
- **Watch View pane**—Displays the values of the watch expressions you enter. The sequence editor updates the values in the Watch View pane when execution suspends at a breakpoint. If you enabled tracing, the sequence editor also updates the values after executing each step and highlights values that change in red. Refer to the *NI TestStand Help* for more information about the Watch View pane.
- **Output pane**—Displays generic messages, warnings, and error messages. By default, the Output pane is empty. Use the `OutputMessage` expression function or the `Engine.NewOutputMessage` method and the `OutputMessage.Post` method to generate the messages you want to display. Each message specifies a severity and a timestamp. The message can also specify an icon, a category, and additional execution information. Double-click a message or right-click a message and select **Goto Location In»Step** from the context menu to go to the step that generates the message. By default, the sequence editor generates output messages for any information an SCC provider generates. Refer to the *NI TestStand Help* for more information about the Output pane and the Output pane context menu.

Terminating and Aborting Executions

The menus in the sequence editor and user interfaces include commands to stop execution before the execution completes normally. The TestStand API includes corresponding methods to stop execution from inside a code module or to determine if the execution stopped. You can issue a stop request at any time to stop one execution or all executions. Stop requests do not take effect in each execution until the currently executing code module returns control.

You can terminate an execution, or you can abort an execution.

When you terminate an execution, all the cleanup steps in the sequences on the call stack run before execution ends. Also, if you terminate an execution while the client sequence file is still running, the process model continues to run, possibly testing the next UUT or generating a test report.

When you abort an execution, the cleanup steps do not run, and the process model does not continue. Abort an execution in cases when you want an execution to completely stop as soon as possible. In general, it is better to terminate execution so the cleanup steps can return the system to a known state. Abort an execution only when you are debugging or when you are sure it is safe to skip the cleanup steps for a sequence.

Result Collection

TestStand automatically collects the results of each step. You can configure result collection for each step on the Run Options panel of the Step Settings pane. You can disable result collection for an entire sequence in the Sequence Properties dialog box or completely disable result collection on a computer in the Station Options dialog box.

Each sequence includes a `ResultList` local variable, which is an empty array of container properties. TestStand appends a new container property, the step result, to the end of the `ResultList` array before a step executes. After the step executes, TestStand copies the contents of the `Result` subproperty for the step into the step result in the `ResultList` array.

Each step type can define different contents for its `Result` subproperty, and TestStand can append step results that contain `Result` properties from different step types to the same `ResultList` array. When TestStand copies the `Result` property for a step to the step result, TestStand also adds the name of the step, its position in the sequence, and other identifying information. For a step that calls a subsequence, TestStand also adds the `ResultList` array variable from the subsequence.

Using the TestStand API, a process model can request that TestStand insert additional step properties in the step results for all steps automatically. A code module can also use the TestStand API to insert additional step result information for a particular step.

Refer to the *NI TestStand Help* for more information about the Step Settings pane, Sequence Properties dialog box, and the Station Options dialog box.

Custom Result Properties

Because each step type can have a different set of subproperties under its Result property, the step result varies according to the step type. Table 3-1 lists the custom properties the step result can contain for steps that use one of the built-in step types.

Table 3-1. Custom Properties in the Step Results for Steps that Use the Built-In Step Types

Custom Step Property	Step Types that Use the Property
Error.Code	All
Error.Msg	All
Error.Occurred	All
Status	All
Common	All
Numeric	Numeric Limit Test, Multiple Numeric Limit Test
PassFail	Pass Fail Test
Limits.String	String Value Test Refer to the Exceptions section in this chapter for more information.
ButtonHit	Message Popup
Response	Message Popup
ExitCode	Call Executable
NumPropertiesRead	Property Loader
NumPropertiesApplied	Property Loader
ReportText	All
Limits.Low	Numeric Limit Test, Multiple Numeric Limit Test, String Value Test Refer to the Exceptions section in this chapter for more information.

Table 3-1. Custom Properties in the Step Results for Steps that Use the Built-In Step Types (Continued)

Custom Step Property	Step Types that Use the Property
Limits.High	Numeric Limit Test, Multiple Numeric Limit Test Refer to the Exceptions section in this chapter for more information.
Comp	Numeric Limit Test, Multiple Numeric Limit Test Refer to the Exceptions section in this chapter for more information.
Measurement	Multiple Numeric Limit Test
AsyncID	Sequence Call Refer to the Exceptions section in this chapter for more information.
AsyncMode	Sequence Call Refer to the Exceptions section in this chapter for more information.



Note Table 3-1 does not include the result properties for Synchronization, Database, IVI, or LabVIEW Utility step types. Refer to Appendix B, [Synchronization Step Types](#), Appendix C, [Database Step Types](#), Appendix D, [IVI Step Types](#), and Appendix E, [LabVIEW Utility Step Types](#), for more information about these step types.

The Common property uses the CommonResults custom data type and is a subproperty of the Result property for every step type. Consequently, you can add a subproperty to the result of every step type by adding a subproperty to the definition of the CommonResults custom data type. However, if you modify the CommonResults custom data type without incrementing the type version number, you might see a type conflict when you open other sequence files, such as `FrontEndCallbacks.seq` when you log in or out. TestStand prompts you to increment the version number when you save changes to any data type or step type. National Instruments recommends modifying the CommonResults data type only if you want to make an architectural change to all step types that you use. Share the modified CommonResults data type and the step types that use the CommonResults data type only with systems on which you are certain no conflicting changes to CommonResults will be deployed.

Exceptions

The Limits.Low, Limits.High, Limits.String, and Comp properties are not subproperties of the Result property for the Numeric Limit Test and the String Value Test step types. Therefore, TestStand does not automatically include these properties in the step result. Depending on options you set during the step configuration, the default process model uses the TestStand API to include the Limits.Low, Limits.High, Limits.String, and Comp properties in the step results for Numeric Limit Test and String Value Test steps that contain these properties.

The AsyncID and AsyncMode properties are not subproperties of the Result property for the Sequence Call step type. TestStand adds these properties to the step results only for Sequence Call steps that call subsequences asynchronously.

Standard Result Properties

In addition to copying step properties to step results, TestStand also adds a set of standard properties to each step result as subproperties of the TS property, as shown in Table 3-2.

Table 3-2. Standard Step Result Properties

Standard Result Property	Description
TS.StartTime	Time at which the step began executing; specifically, the number of seconds since the TestStand Engine initialized.
TS.TotalTime	Number of seconds the step took to execute; includes the time for all step options, including preconditions, expressions, post actions, module loading, and module execution.
TS.ModuleTime	Number of seconds the code module took to execute.
TS.Index	Zero-based position of the step in the step group.
TS.StepName	Name of the step.
TS.StepGroup	Step group that contains the step. The values are Main, Setup, or Cleanup.
TS.StepId	Unique Step Id, which is a GUID represented as a string that begins with “ID#.” and contains 26 characters (only alphanumeric characters and the special characters “#,” “.”, “+,” and “/”). TestStand attempts to maintain globally unique step Ids, but copying files on disk does not prevent duplicate Ids.

Table 3-2. Standard Step Result Properties (Continued)

Standard Result Property	Description
TS.Id	A number TestStand assigns to the step result. The number is unique with respect to all other step results in the current TestStand session.
TS.InteractiveExeNum	A number TestStand assigns to an interactive execution. The number is unique with respect to all other interactive executions in the current TestStand session. TestStand adds this property only if you run the step interactively.
TS.StepType	Name of the step type.
TS.Server	The name of the server computer on which the step runs the subsequence it calls. TestStand adds this property only for Sequence Call steps that run subsequences on a remote computer.
TS.StepCausedSequenceFailure	TestStand adds this property only if the step fails. The value is <code>True</code> if the step failure causes the sequence to fail. The value is <code>False</code> if the step failure does not cause the sequence to fail or if the sequence has already failed.
TS.BlockLevel	Indicates the number of blocks that encloses the step, such as If and For steps. The value is zero for top-level steps.

Subsequence Results

If a step calls a subsequence or generates a call to a callback sequence, TestStand creates a special step result subproperty to store the result of the subsequence unless the callback or sequence disables results. Table 3-3 lists the name of the subproperty for each type of subsequence call.

Table 3-3. Subproperty Names for Subsequence Results

Result Subproperty Name	Type of Subsequence Call
TS.SequenceCall	Sequence Call
TS.PostAction	Post Action callback
TS.SequenceFilePreStep	SequenceFilePreStep callback
TS.SequenceFilePostStep	SequenceFilePostStep callback
TS.ProcessModelPreStep	ProcessModelPreStep callback
TS.ProcessModelPostStep	ProcessModelPostStep callback

Table 3-3. Subproperty Names for Subsequence Results (Continued)

Result Subproperty Name	Type of Subsequence Call
TS.StationPreStep	StationPreStep callback
TS.StationPostStep	StationPostStep callback
TS.SequenceFilePreInteractive	SequenceFilePreInteractive callback
TS.SequenceFilePostInteractive	SequenceFilePostInteractive callback
TS.ProcessModelPreInteractive	ProcessModelPreInteractive callback
TS.ProcessModelPostInteractive	ProcessModelPostInteractive callback
TS.StationPreInteractive	StationPreInteractive callback
TS.StationPostInteractive	StationPostInteractive callback
TS.SequenceFilePostResultListEntry	SequenceFilePostResultListEntry callback
TS.ProcessModelPostResultListEntry	ProcessModelPostResultListEntry callback
TS.StationPostResultListEntry	StationPostResultListEntry callback
TS.SequenceFilePostStepRuntimeError	SequenceFilePostStepRuntimeError callback
TS.ProcessModelPostStepRuntimeError	ProcessModelPostStepRuntimeError callback
TS.StationPostStepRuntimeError	StationPostStepRuntimeError callback
TS.SequenceFilePostStepFailure	SequenceFilePostFailure callback
TS.ProcessModelPostStepFailure	ProcessModelPostFailure callback
TS.StationPostStepFailure	StationFilePostFailure callback

TestStand adds the following properties to the subproperty for each subsequence:

- **SequenceFile**—Absolute path of the sequence file that contains the subsequence.
- **Sequence**—Name of the subsequence the step called.
- **Status**—Status of the subsequence the step called.
- **ResultList**—Value of Locals.ResultList for the subsequence the step called. This property contains the results for the steps in the subsequence.

Loop Results

When you configure a step to loop, you can use the Record Result of Each Iteration option on the Looping panel of the Step Settings pane to direct TestStand to store a separate result for each loop iteration in the result list. In the result list, the results for the loop iterations immediately follow the result for the step as a whole.

TestStand adds a `TS.LoopIndex` numeric property to each loop iteration result to record the value of the loop index for the iteration. TestStand also adds the following special loop result properties to the main result for the step:

- **TS.EndingLoopIndex**—Value of the loop index when looping completes.
- **TS.NumLoops**—Number of times the step loops.
- **TS.NumPassed**—Number of loops for which the step status is Passed or Done.
- **TS.NumFailed**—Number of loops for which the step status is Failed.

Report Generation

When you run a sequence using the Test UUTs or Single Pass Execution entry point, the default process model generates the test report by traversing the results for the Main sequence in the client sequence file and all the subsequences it calls.

Refer to the *Process Models* section of Chapter 1, *NI TestStand Architecture*, Chapter 10, *Customizing Process Models and Callbacks*, and Appendix A, *Process Model Architecture*, for more information about process models. Refer to Chapter 6, *Database Logging and Report Generation*, for more information about report generation.

Engine Callbacks

TestStand specifies a set of Engine callback sequences it invokes at specific points during execution.

Use Engine callbacks to make TestStand call certain sequences before and after the execution of individual steps, before and after interactive executions, after loading a sequence file, and before unloading a sequence

file. Because the TestStand Engine controls the execution of steps and the loading and unloading of sequence files, TestStand defines the set of Engine callbacks and their names.

Refer to Chapter 10, *Customizing Process Models and Callbacks*, for more information about Engine callbacks.

Step Execution

Depending on the options you set during step configuration, a step performs a number of actions as it executes. Table 3-4 lists the most common actions a step can take, in the order the step performs them.

Table 3-4. Order of Actions a Step Performs

Action Number	Description	Remarks
1	Allocate step result	—
2	Enter batch synchronization section	If option is set
3	Evaluate precondition	If <code>False</code> , perform Action Number 25, then proceed to Action Number 29
4	Acquire step lock	If option is set
5	Check run mode	—
6	Load module if not already loaded	—
7	Execute step switching	—
8	Evaluate Loop Initialization expression	Only if looping
9	Evaluate Loop While expression, skip to Action Number 23 if <code>False</code>	Only if looping
10	Allocate loop iteration result	Only if looping
11	Call Pre-Step Engine callbacks	—
12	Evaluate Pre-Expression	—
13	Call Pre-Step substeps for step type	—

Table 3-4. Order of Actions a Step Performs (Continued)

Action Number	Description	Remarks
14	Call module	—
15	Call Post-Step substeps for step type	TestStand calls Post-Step substeps even if the user code module generates a run-time error, which enables Post-Step substeps to perform error handling, if appropriate.
16	Evaluate Post-Expression	—
17	Evaluate Status expression	—
18	Call Post-Step Engine callbacks	—
19	Call Post-Step Failure Engine callback	Only if loop iteration fails
20	Fill out loop iteration result	Only if looping
21	Call Post-ResultList Entry Engine callback	Only if looping
22	Evaluate Loop Increment expression, return to Action Number 9	Only if looping
23	Evaluate Loop Status expression	Only if looping
24	Disconnect switching routes with step lifetime	—
25	Unload module if required	—
26	Update sequence failed state	
27	Call Post-Step Failure Engine callback	Only if step fails
28	Execute post action	—
29	Release step lock	If option is set
30	Exit batch synchronization section	If option is set

Table 3-4. Order of Actions a Step Performs (Continued)

Action Number	Description	Remarks
31	Fill out step result	—
32	Call Post-ResultList Entry Engine callback	—

Usually, a step performs only a subset of these actions depending on the configuration of the step and the test station. When TestStand detects a run-time error, it calls the Post-Step Error Engine callbacks. If you do not define these callbacks or if the callbacks do not reset the error state for the step, TestStand unloads the module and then releases the step lock. If a run-time error occurs in a loop iteration, TestStand fills out the loop iteration result before unloading the module and releasing the step lock.

Step Status

Every step has a `Result.Status` property, which is a string that indicates the result of the step execution. Although TestStand imposes no restrictions on the values to which the step or its code module can set the status property, TestStand and the built-in step types use and recognize the values that appear in Table 3-5.

Table 3-5. Standard Values for the Status Property

String Value	Meaning	Source of the Status Value
Passed	Indicates the step performed a test that passed.	Step or code module
Failed	Indicates the step performed a test that failed.	Step or code module
Error	Indicates a run-time error occurred.	TestStand
Done	Indicates the step completed without setting its status.	TestStand

Table 3-5. Standard Values for the Status Property (Continued)

String Value	Meaning	Source of the Status Value
Terminated	Indicates the step did not set the status and a request to terminate the execution occurred while executing the step. When TestStand returns a status of <code>Terminated</code> to a calling sequence, TestStand sets the step status to <code>Terminated</code> . If you enabled the Ignore Termination option on the Run Options panel of the Step Settings pane for a Sequence Call step, TestStand does not return the request to terminate the execution to the calling sequence invocation. The status of the execution is <code>Terminated</code> if TestStand returns the request to terminate the execution to the root sequence invocation.	TestStand
Skipped	Indicates the step did not execute because the run mode for the step is <code>Skip</code> .	TestStand
Running	Indicates the step is currently running.	TestStand
Looping	Indicates the step is currently running in loop mode.	TestStand

Failures

If you enable the Step Failure Causes Sequence Failure option on the Run Options panel of the Step Settings pane, TestStand sets the sequence status to `Failed` when a step fails. When the sequence returns as `Failed`, the Sequence Call step also fails. In this way, a step failure in a subsequence can propagate up through the chain of Sequence Call steps. By default, TestStand enables the Step Failure Causes Sequence Failure option for most step types.

You can also control how execution proceeds after a step failure causes a sequence to fail. To configure an execution to jump to the Cleanup step group upon failure, enable the Immediately Goto Cleanup on Sequence Failure option in the Sequence Properties dialog box. By default, TestStand disables this option.

Terminations

When you request to terminate an execution, the currently executing sequence invocation for each thread immediately calls the Cleanup step group. When a terminating subsequence returns to a calling sequence, TestStand sets the calling sequence step status to `Terminated`, and the calling sequence continues the termination process down the call stack unless you enabled the Ignore Termination option on the Run Options panel of the Step Settings pane for the Sequence Call step. If you enabled this setting, TestStand ignores the termination of the execution for the thread, and the thread execution continues normally. If TestStand returns the request to terminate the execution to the root sequence invocation, the result status for the execution is `Terminated`.

Run-Time Errors

TestStand generates a run-time error if it encounters a condition that prevents a sequence from executing. For example, if a precondition refers to the status of a step that does not exist, TestStand generates a run-time error when it attempts to evaluate the precondition. TestStand also generates a run-time error when a code module causes an access violation or any other exception.

TestStand does not use run-time errors to indicate UUT test failures. Instead, a run-time error indicates a problem exists with the testing process itself and testing cannot continue. Usually, a code module reports a run-time error if it detects an error in a hardware or software resource it uses to perform a test.

TestStand allows you to decide interactively how to handle a run-time error. To interactively handle a run-time error, configure TestStand to launch the Run-Time Error dialog box in the event of an error by selecting **Show Dialog** from the On Run-Time Error ring control on the **Execution** tab of the Station Options dialog box. Refer to the *NI TestStand Help* for more information about the Station Options and Run-Time Error dialog boxes.

TestStand also allows you to invoke Post-Step RunTime Error Engine callbacks when a run-time error occurs. Refer to Chapter 10, [Customizing Process Models and Callbacks](#), for more information about Engine callbacks.

Built-In Step Types

TestStand groups the core set of built-in step types into the following categories:

- Step types you can use with any module adapter. Step types such as the Numeric Limit Test and the String Value Test call any code module you specify. They also might perform additional actions, such as comparing a value the code module returns with limits you specify.
- Step types that always use a specific module adapter to call code modules. Sequence Call is the only step type in this category.
- Step types that perform a specific action and do not require you to specify a code module. Step types such as Message Popup, Statement, and Flow Control perform actions you configure in an edit tab or edit dialog box specific to the step type.



Note TestStand also includes sets of application-specific step types. For example, TestStand provides sets of step types that make it easier to synchronize multiple threads, access databases, control IVI instruments, and access VIs and remote systems. Refer to Appendix B, [Synchronization Step Types](#), Appendix C, [Database Step Types](#), Appendix D, [IVI Step Types](#), and Appendix E, [LabVIEW Utility Step Types](#), for more information about these step types.

Using Step Types

Use step types when you insert steps in the Setup, Main, and Cleanup groups of the Steps pane in the Sequence File window. You can insert a step using the Step Types list in the Insertion Palette, shown in Figure 4-1, or the Insert Step submenu in the Steps pane context menu. The Insertion Palette and the Insert Step submenu list all the available step types. Refer to the *NI TestStand Help* for more information about the Insertion Palette.

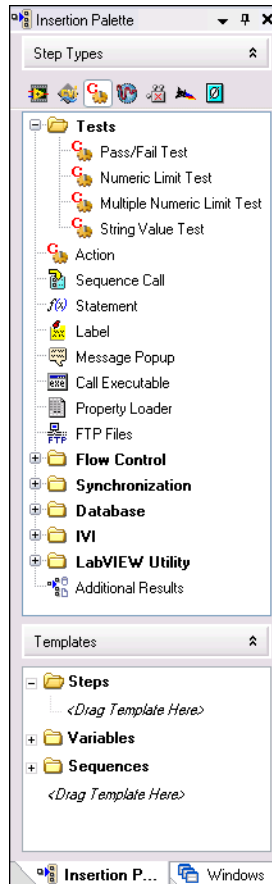


Figure 4-1. Insertion Palette

When you insert a step type from the Insertion Palette or the Insert Step submenu, TestStand creates a step using the step type and the module adapter selected in the Insertion Palette or toolbar. After you insert the step, select Specify Module from the context menu to specify the code module or sequence, if any, the step calls. The Specify Module command displays a Module tab on the Step Settings pane that is different for each adapter. Refer to Chapter 5, *Module Adapters*, and the *NI TestStand Help* for more information about the Module tab for each adapter.

For each step type, other items appear in the context menu above the Specify Module item. For example, the Edit Limits item appears in the context menu for Numeric Limit Test steps, and the Edit Data Source item appears in the context menu for Pass/Fail Test steps. Select the menu item to display a step-type-specific pane or launch a step-type-specific dialog

box in which you can modify step properties specific to the step type. Refer to the *NI TestStand Help* for more information about the menu items for each of the built-in step types.

To modify step properties common to all step types, select the Properties tab on the Step Settings pane. Refer to the *NI TestStand Help* for more information about the Step Settings pane.

The Insertion Palette also contains a Templates list you can use to hold copies of sequences, steps, and variables you reuse. Refer to the *NI TestStand Help* for more information about the Templates list of the Insertion Palette.

Built-In Step Properties

TestStand steps include built-in properties you can specify using the following panels on the Properties tab of the Step Settings pane:

General Panel

- **Name**—Specifies the name of the step.
- **Type**—Specifies the type of the step.
- **Adapter**—Specifies the adapter the step uses to call a code module.
- **Icon**—Specifies the icon to display for the step.
- **Comment**—Specifies the comment of the step.

Run Options Panel

- **Load/Unload Options**—Specifies how TestStand loads and unloads the code modules or subsequences each step invokes.
- **Run Mode**—Specifies if TestStand skips a step or forces the step to pass or fail without executing the code module of the step.
- **Precondition Evaluation in Interactive Mode**—Specifies if TestStand evaluates the step precondition when you run the step interactively.
- **TestStand Window Activation**—Specifies if the TestStand application activates its window when the step completes.
- **Sequence Call Trace Setting**—Specifies if TestStand traces the steps in the subsequence the step calls. This property exists only for Sequence Call steps.

- **Record Result**—Specifies if TestStand collects the results for this step. Refer to the [Result Collection](#) section of Chapter 3, *Executions*, for more information about result collection.
- **Step Failure Causes Sequence Failure**—Specifies if TestStand sets the status of the sequence to `Failed` when the status of the step is `Failed`.
- **Ignore Run-Time Errors**—Specifies if TestStand continues execution normally after the step when a run-time error occurs in the step.



Caution Some run-time errors can place the system in a bad state, and continuing with the execution can result in an undefined behavior.

- **Ignore Termination**—Specifies if a Sequence Call step ignores the request to terminate execution.

Looping Panel

- **Loop**—Specifies if the step executes once or multiple times before executing the next step. You can specify the conditions under which to terminate the loop. You can also specify to collect results for each loop iteration, for the loop as a whole, or for both.

Post Actions Panel

- **Post Actions**—Specifies if TestStand executes a specific sequence or jumps to other steps after executing the step, depending on the pass/fail status of the step or any custom condition.

Switching Panel

- **Switching**—Specifies if TestStand performs any switching operations when the step executes.

Synchronization Panel

- **Synchronization**—Specifies if a step should block another instance of the step from executing at the same time in a different thread.

Expressions Panel

- **Pre-Expressions**—Specifies an expression to evaluate before executing the code module of the step.
- **Post-Expressions**—Specifies an expression to evaluate after executing the code module of the step.
- **Status Expression**—Specifies an expression that determines the value of the status property of the step.

Preconditions Panel

Specifies the conditions that must be `True` for TestStand to execute the step during the normal flow of execution in a sequence.

Requirements Panel

Notates product and unit requirements a step covers.

Additional Results Panel

Allows you to add and configure additional results. An additional result is a value TestStand adds to the result list of a step when the step executes. An additional result can be a module parameter or a custom additional result in which you specify the name and value of the result.

Property Browser Panel

Displays the built-in and custom properties for a step.

Custom Step Properties

You can usually modify the values of custom step properties using the tabs on the Step Settings pane. If the step type does not include a tab for the custom properties, select the Property Browser panel to view the custom properties for the step. Although code modules usually do not modify the values of the built-in step properties at run time, they often modify and read the values of the custom step properties when determining the pass/fail status.

Refer to the *NI TestStand Help* for more information about the Properties tab of the Step Settings pane.

Custom Properties All Step Types Share

Each step type defines its own set of custom properties. All steps that use the same step type have the same set of custom properties.

All built-in step types contain the following custom properties:

- **Step.Result.Error.Code**—Code that describes the error that occurred.
- **Step.Result.Error.Msg**—Message string that describes the error that occurred.
- **Step.Result.Error.Occurred**—Boolean flag that indicates if a run-time error occurred in the step. TestStand documentation refers to this property as the error occurred flag.

The error occurred flag can become `True` when a run-time error condition occurs and the code module or module adapter sets the value to `True` or when an unhandled exception occurs in the code module or at any other time during step execution. When a step finishes execution and the error occurred flag is `True`, the TestStand Engine responds in the following ways:

- Makes no evaluation of status and post-expressions for a step and sets the step status to `Error`.
- Evaluates the Ignore Run-Time Errors step property.
 - If `False`, TestStand reports the run-time error to the sequence.
 - If `True`, TestStand continues execution normally after the step.
- **Step.Result.Status**—Specifies the status of the last execution of the step, such as `Done`, `Passed`, `Failed`, `Skipped`, or `Error`. TestStand documentation refers to this property as the step status.

Before TestStand executes a step, it sets the step status to `Running` or `Looping`. When a step finishes execution and the error occurred flag is `False`, TestStand changes the step status to `Done`. The step status becomes `Passed` or `Failed` only when a code module, module adapter, or step type explicitly sets the step status to `Passed` or `Failed`. Refer to Chapter 5, [Module Adapters](#), for more information about the assignments module adapters make to and from step properties.

- **Step.Result.ReportText**—Contains a message string TestStand includes in the report.
- **Step.Result.Common**—Placeholder container you can customize by modifying the CommonResults standard data type. Refer to the [Using Data Types](#) section of Chapter 12, *Standard and Custom Data Types*, for more information about standard TestStand data types.

Step Types You Can Use with Any Module Adapter

TestStand includes five built-in step types you can use with any module adapter—Pass/Fail Test, Numeric Limit Test, Multiple Numeric Limit Test, String Value Test, and Action. When you insert a step in a sequence, you must select a module adapter in the Step Types list of the Insertion Palette or from the Adapter ring control located on the sequence editor toolbar. TestStand assigns the adapter you selected when you insert the step. Once you add a step, you can change the adapter associated with the selected step on the General panel on the Step Settings pane.

TestStand uses the following adapter icons for each step:



LabVIEW Adapter



LabWindows/CVI Adapter



C/C++ DLL Adapter



.NET Adapter



ActiveX/COM Adapter



HTBasic Adapter



Sequence Adapter



<None>

If you choose the <None> adapter, the step does not call a code module.

To specify the code module the step calls, select **Specify Module** from the step context menu or click the **Module** tab on the Step Settings pane. Refer to the *NI TestStand Help* for more information about the Module tab for each module adapter.

Pass/Fail Test

Use a Pass/Fail Test step to call a code module that makes its own pass/fail determination. After the code module executes, the Pass/Fail Test step type evaluates the `Step.Result.PassFail` property. If `Step.Result.PassFail` is `True`, the step type sets the step status to `Passed`. If `Step.Result.PassFail` is `False`, the step type sets the step status to `Failed`.

A code module can set the value of `Step.Result.PassFail` in the following ways:

- **LabVIEW Adapter**—Specify `Step.Result.PassFail` as the Value expression for a Boolean output of a VI on the LabVIEW Module tab.
- **LabWindows/CVI, C/C++ DLL, .NET, ActiveX/COM, or Sequence Adapter**—Pass `Step.Result.PassFail` as a reference parameter to a subsequence or code module.
- **LabVIEW or LabWindows/CVI Adapter**—The LabVIEW and LabWindows/CVI Adapters update the value of `Step.Result.PassFail` automatically after calling legacy code modules. The LabVIEW Adapter updates the value of `Step.Result.PassFail` based on the value of the Pass/Fail Flag element of the **Test Data** cluster the VI returns. The LabWindows/CVI Adapter updates the value of `Step.Result.PassFail` based on the value of the result field of the **tTestData** parameter it passes to the C function.

Refer to the *Using LabVIEW with TestStand* manual and the *Using LabWindows/CVI with TestStand* manual for more information about the assignments the module adapters automatically make to and from step properties for legacy code modules in LabVIEW and LabWindows/CVI.

- **All Adapters**—Use the TestStand API to set the value of `Step.Result.PassFail` directly in a code module.

By default, the step type uses the value of the `Step.Result.PassFail` Boolean property to determine if the step passes or fails. To customize the Boolean expression that determines if the step passes, select **Edit Data Source** from the context menu for the step or click the **Data Source** tab of the Step Settings pane.

Refer to the *NI TestStand Help* for more information about the Pass/Fail Test step edit tabs.

In addition to the common custom properties, the Pass/Fail Test step type defines the following step properties:

- **Step.Result.PassFail**—Specifies the Boolean pass/fail flag. Pass is `True`. Fail is `False`. Usually, you set this value in the code module or with a custom pass/fail source expression.
- **Step.InBuf**—Specifies an arbitrary string the LabVIEW and LabWindows/CVI Adapters pass to the test in the **Input Buffer** control or **tTestData** structure of legacy code modules.

This property exists to maintain compatibility with previous test executives. Usually, code modules you develop for TestStand receive data as input parameters or access data as properties using the TestStand API.

- **Step.DataSource**—Specifies the Boolean expression the step uses to set the value of `Step.Result.PassFail`. The default value of the expression is `"Step.Result.PassFail"`, which has the effect of using the value the code module sets. You can customize this expression if you do not want to set the value of `Step.Result.PassFail` in the code module. For example, you can set the data source expression to refer to multiple variables and properties, such as `RunState.PreviousStep.Result.Numeric * Locals.Attenuation > 12`.

Numeric Limit Test

Use a Numeric Limit Test step to call a code module that returns a single measurement value. After the code module executes, the Numeric Limit Test step type compares the measurement value to predefined limits. If the measurement value is within the bounds of the limits, the step type sets the step status to `Passed`. Otherwise, the step type sets the step status to `Failed`.

A Numeric Limit Test step uses the `Step.Result.Numeric` property to store the measurement value. A code module can set the value of `Step.Result.Numeric` in the following ways:

- **LabVIEW Adapter**—Specify `Step.Result.Numeric` as the Value expression for a Numeric output of a VI on the LabVIEW Module tab.
- **LabWindows/CVI, C/C++ DLL, .NET, ActiveX/COM, or Sequence Adapter**—Pass `Step.Result.Numeric` as a reference parameter to a subsequence or code module.
- **LabVIEW or LabWindows/CVI Adapter**—The LabVIEW and LabWindows/CVI Adapters update the value of `Step.Result.Numeric` automatically after calling legacy code modules. The LabVIEW

Adapter updates the value of `Step.Result.Numeric` based on the value of the Numeric Measurement element of the **Test Data** cluster the VI returns. The LabWindows/CVI Adapter updates the value of `Step.Result.Numeric` based on the value of the measurement field of the **tTestData** parameter it passes to the C function.

Refer to the *Using LabVIEW with TestStand* manual and the *Using LabWindows/CVI with TestStand* manual for more information about the assignments the module adapters automatically make to and from step properties for legacy code modules in LabVIEW and LabWindows/CVI.

- **All Adapters**—Use the TestStand API to set the value of `Step.Result.Numeric` directly in a code module.

Refer to the *NI TestStand Help* for more information about the Numeric Limit Test step edit tab.

By default, the step type uses the value of the `Step.Result.Numeric` property as the numeric measurement to compare the limits against.

In addition to the common custom properties, the Numeric Limit Test step type defines the following step properties:

- **Step.Result.Numeric**—Specifies the numeric measurement value. Usually, you set this value in the code module.
- **Step.Result.Units**—Specifies a label that indicates the unit of measurement.
- **Step.Limits.Low, High, LowExpr, HighExpr, UseLowExpr, and UseHighExpr**—Specify the limits for the comparison.
- **Step.Comp**—Specifies the type of comparison, such as EQ.
- **Step.CompExpr**—Specifies the comparison operation using an expression.
- **Step.UseCompExpr**—Specifies to use the expression to compare the measurement values.
- **Step.InBuf**—Specifies an arbitrary string the LabVIEW and LabWindows/CVI Adapters pass to the test in the **Input Buffer** control or **tTestData** structure of legacy code modules.

This property exists to maintain compatibility with previous test executives. Usually, code modules you develop for TestStand receive data as input parameters or access data as properties using the TestStand API.

- **Step.DataSource**—Specifies a numeric expression the step type uses to set the value of `Step.Result.Numeric`. The default value of the expression is `"Step.Result.Numeric"`, which has the effect of using the value the code module sets. You can customize this expression if you do not want to set the value of `Step.Result.Numeric` in the code module.

You can use a Numeric Limit Test step without a code module, which is useful when you want to limit-check a value you have already acquired. To set up this limit check, select **<None>** as the module adapter before you insert the step in the sequence and configure `Step.DataSource` to specify the value you have already acquired.

Multiple Numeric Limit Test

Use a Multiple Numeric Limit Test step to limit-check a set of related measurements. Although you can use several Numeric Limit Test steps to limit test a set of related measurements, using the Multiple Numeric Limit Test step type to check limits for multiple measurements in a single step might be easier.

You can test limits for any number of measurements with the Multiple Numeric Limit Test step. Each measurement can have independent limits, units, display formats, data sources, and comparison types. A Multiple Numeric Limit Test step passes if all its measurements pass. Configure each measurement the same way you configure an individual Numeric Limit Test step.

Refer to the *NI TestStand Help* for more information about the Multiple Numeric Limit Test step edit tabs.

In addition to the common custom properties, the Multiple Numeric Limit Test step type defines the following step properties:

- **Step.Result.Measurement**—An array that stores the measurements you configure for the step. Each element of the measurement array is an instance of the `NI_LimitMeasurement` data type. The `NI_LimitMeasurement` type defines the following fields:
 - **Limits.Low, High, LowExpr, HighExpr, UseLowExpr, and UseHighExpr**—Specify the limits for the comparison.
 - **Units**—Specifies a label that describes the measurement units for the limits and the measurement value.
 - **Comp**—Specifies the type of comparison, such as EQ.

- **CompExpr**—Specifies the comparison operation using an expression.
- **UseCompExpr**—Specifies to use the expression to compare the measurement values.
- **Data**—Stores the numeric measurement value. The step obtains this value from the corresponding element in Step.NumericArray or from the data source you specify.
- **Status**—Stores the result of the comparison of the measurement value with the limits. The result is `Passed` or `Failed`.
- **Step.DataSource**—Specifies an expression that identifies the numeric array that provides the data values for all measurements when you do not use a separate data source for each measurement.
- **Step.NumericArray**—Provides a numeric array that is the default data source Step.DataSource specifies.
- **Step.UseIndividualDataSources**—Specifies if the step stores separate data source expressions for each measurement in the Step.DataSourceArray. If this property is `False`, the step obtains the data values for each measurement from the numeric array the Step.DataSource property specifies.
- **Step.DataSourceArray**—Specifies a data source for each measurement element in the measurement array.
- **Step.ExpectedNumMeasure**—Specifies the number of measurements for the step.
- **Step.ExtraDataAction**—Specifies how the step processes data if the numeric array contains more elements than the number of measurements. The step can apply a specific measurement to extra data, repeat the measurement set again, generate a run-time error, or ignore the extra data.
- **Step.MeasToRepeat**—Specifies a measurement to repeat when the Step.ExtraDataAction is set to `RepeatOne`.
- **Step.ExtraMeasAction**—Specifies if the step ignores, takes no action, or generates a run-time error when the numeric array contains fewer elements than the expected number of measurements.

String Value Test

Use a String Value Test step to call a code module that returns a string value. After the code module executes, the String Value Test step type compares the string the step obtains to the string the step expects to receive. If the string the step obtains matches the string it expects, the step type sets the step status to `Passed`. Otherwise, the step type sets the step status to `Failed`.

A String Value Test step uses the `Step.Result.String` property to store the string value. A code module can set the value of `Step.Result.String` in the following ways:

- **LabVIEW Adapter**—Specify `Step.Result.String` as the Value expression for a Numeric output of a VI on the LabVIEW Module tab.
- **LabWindows/CVI, C/C++ DLL, .NET, ActiveX/COM, or Sequence Adapter**—Pass `Step.Result.String` as a reference parameter to a subsequence or code module.
- **LabVIEW or LabWindows/CVI Adapter**—The LabVIEW and LabWindows/CVI Adapters update the value of `Step.Result.String` automatically after calling legacy code modules. The LabVIEW Adapter updates the value of `Step.Result.String` based on the value of the String Measurement element of the **Test Data** cluster the VI returns. The LabWindows/CVI Adapter updates the value of `Step.Result.String` based on the value of the `stringMeasurement` field of the **tTestData** parameter it passes to the C function.

Refer to the *Using LabVIEW with TestStand* manual and the *Using LabWindows/CVI with TestStand* manual for more information about the assignments the module adapters automatically make to and from step properties for legacy code modules in LabVIEW and LabWindows/CVI.

- **All Adapters**—Use the TestStand API to set the value of `Step.Result.String` directly in a code module.

By default, the step type uses the value of the `Step.Result.String` property as the string value to compare the limits against.

Refer to the *NI TestStand Help* for more information about the String Value Test step edit tabs.

In addition to the common custom properties, the String Value Test step type defines the following step properties:

- **Step.Result.String**—Specifies the string value. Usually, you set this value in the code module.
- **Step.Limits.String**, **StringExpr**, and **UseStringExpr**—Specifies the expected string for the string comparison.
- **Step.Comp**—Specifies the type of comparison, such as Ignore Case.
- **Step.CompExpr**—Specifies the comparison operation using an expression.
- **Step.UseCompExpr**—Specifies to use the expression to compare the string values.
- **Step.InBuf**—Specifies an arbitrary string the LabVIEW and LabWindows/CVI Adapters pass to the test in the **Input Buffer** control or **tTestData** structure of legacy code modules.

This property exists to maintain compatibility with previous test executives. Usually, code modules you develop for TestStand receive data as input parameters or access data as properties using the TestStand API.

- **Step.DataSource**—Specifies a string expression the step type uses to set the value of Step.Result.String. The default value of the expression is `Step.Result.String`, which has the effect of using the value that the code module sets. You can customize this expression if you do not want to set the value of Step.Result.String in the code module.

You can use a String Value Test step without a code module, which is useful when you want to test a string you have already acquired. To set up this test, select **<None>** as the module adapter before you insert the step in the sequence and configure `Step.DataSource` to specify the string you have already acquired.

Action

Use Action steps to call code modules that do not perform tests but perform actions necessary for testing, such as initializing an instrument. By default, Action steps do not pass or fail. The step type does not modify the step status. Therefore, the status for an Action step is `Done` or `Error` unless the code module specifically sets another status for the step or the step calls a subsequence that fails. When an action uses the Sequence Adapter to call a subsequence, and the subsequence fails, the Sequence Adapter sets the status of the step to `Failed`.

The Action step type does not define any additional step properties other than the custom properties all steps contain.

Step Types That Work with a Specific Module Adapter



Use a Sequence Call step, shown at left, to call another sequence in the current sequence file or in another sequence file. A Sequence Call step always uses the Sequence Adapter.



Note By default, the Sequence Adapter is hidden in the Adapter ring control. To enable it, select **Configure»Adapters** from the TestStand menu bar and remove the checkmark from the checkbox in the Hidden column.

You can use the Sequence Adapter with other step types, such as the Pass/Fail Test or the Numeric Limit Test. Using a Sequence Call step is the same as using an Action step with the Sequence Adapter except that the Sequence Call step type sets the step status to `Passed` rather than `Done` when the subsequence succeeds. If the sequence fails, the Sequence Adapter sets the Sequence Call step status to `Failed`. A sequence fails if the status for a step in the sequence is `Failed` and you enabled the Step Failure Causes Sequence Failure option on the Run Options panel of the Step Settings pane. If a run-time error occurs in the subsequence, the Sequence Adapter sets the step status to `Error`.

Refer to the *NI TestStand Help* for more information about the Step Settings pane and the Edit Sequence Call dialog box.

The Sequence Call step type does not define any additional step properties other than the custom properties all steps contain.

TestStand adds the following properties to the results for Sequence Call steps in sequences you configured to run in a new thread or execution.

These properties are not subproperties of the Result property for the Sequence Call step type.

- **AsyncMode**—Set to `True` if the Sequence Call step ran the sequence in a new thread. Set to `False` if the Sequence Call step ran the sequence in a new execution.
- **AsyncID**—Contains the value of the ID property of the thread or execution running the sequence.

Step Types That Do Not Use Module Adapters

Some step types do not use module adapters. When you create an instance of one of these step types, you use the edit tabs or dialog boxes, which you access through the context menu of the step or the Step Settings pane, to configure the step. You do not specify a code module.

Flow Control

Use Flow Control steps to control execution flow within a sequence. The Steps pane automatically inserts steps that complete the flow control block, such as inserting a Case and End step when you insert a Select step. The Steps pane also indents flow control blocks and highlights errors in flow control. Refer to the *NI TestStand Help* for more information about the edit tabs for the Flow Control step types.

If



Use If steps, shown at left, to define a block of steps that execute if a condition is met.

In addition to the common custom properties, the If step type defines the following step property:

- **Step.ConditionExpr**—Specifies the expression that must evaluate to `True` for the steps within the If block to execute.

Else



Use Else steps, shown at left, to define a block of steps that execute if the condition defined by the preceding If or Else If step is not met.

The Else step type does not define any additional step properties other than the custom properties all steps contain.

Else If



Use Else If steps, shown at left, to define a block of steps that execute if a condition is met and the conditions defined by the preceding If step and any preceding Else If step are not met.

In addition to the common custom properties, the Else If step type defines the following step property:

- **Step.ConditionExpr**—Specifies the expression that must evaluate to `True` for the steps within the Else If block to execute.

For



Use For steps, shown at left, to define a block of steps that execute repeatedly for a number of iterations.

In addition to the common custom properties, the For step type defines the following step properties:

- **Step.InitializationExpr**—Specifies the expression the step evaluates before executing the steps within the block the first time. The expression typically initializes a count variable.
- **Step.ConditionExpr**—Specifies the expression that must evaluate to `True` for the steps within the For block to execute.
- **Step.IncrementExpr**—Specifies the expression the step evaluates after each execution of the steps within the block. The expression typically increments a count variable.
- **Step.CustomLoop**—Specifies if the step uses custom expressions to define the looping behavior for the steps within the For block.

For Each



Use For Each steps, shown at left, to define a block of steps that execute once for each element in an array.

In addition to the common custom properties, the For Each step type defines the following step properties:

- **Step.ArrayExpr**—Specifies the expression that determines the array over which the loop iterates.
- **Step.ArrayElementExpr**—Specifies the expression that determines the variable into which to store the current element of the array during each iteration of the loop.
- **Step.OffsetExpr**—Specifies the expression that determines the variable into which to store the current offset of the array during each iteration of the loop.
- **Step.SubscriptExpr**—Specifies the expression that determines the variable into which to store the subscript of the current element in the array during each iteration of the loop.

While



Use While steps, shown at left, to define a block of steps that execute while a condition is `True`.

In addition to the common custom properties, the While step type defines the following step property:

- **Step.CustomExpr**—Specifies the expression the step evaluates before executing the steps within the block.

Do While



Use Do While steps, shown at left, to define a block of steps that execute once and then repeatedly while a condition is `True`.

In addition to the common custom properties, the Do While step type defines the following step property:

- **Step.CustomExpr**—Specifies the expression the step evaluates before executing the steps within the block.

Break



Use a Break step, shown at left, to cause a For, For Each, While, or Do While loop block or a Case block to exit before completing.

The Break step type does not define any additional step properties other than the custom properties all steps contain.

Continue



Use a Continue step, shown at left, to cause the next iteration of an enclosing For, For Each, While, or Do While loop block to begin.

The Continue step type does not define any additional step properties other than the custom properties all steps contain.

Select



Use a Select step, shown at left, to define a block of steps that encloses the sub-blocks a Case step defines. The Select step specifies an expression that determines which Case block executes.

In addition to the common custom properties, the Select step type defines the following step property:

- **Step.ItemExpr**—Specifies the expression that determines which Case block within the Select block executes.

Case



Use a Case step, shown at left, to define a block of steps within a Select block that executes if the expression the Select step specifies evaluates to a certain value.

In addition to the common custom properties, the Case step type defines the following step properties:

- **Step.ItemExpr**—Specifies the expression that determines which Case block within the Select block executes.
- **Step.IsDefault**—Specifies which step defines the default case for the surrounding Select block.

Goto



Use Goto steps, shown at left, to set the next step the TestStand Engine executes. You usually use a Label step as the target of a Goto step, which you can use to rearrange or delete other steps in a sequence without having to change the specification of targets in Goto steps.

Refer to the *NI TestStand Help* for more information about the Destination edit tab.

The Goto step type does not define any additional step properties other than the custom properties all steps contain.

End



Use an End step, shown at left, to define the end of any block of steps.

The End step type does not define any additional step properties other than the custom properties all steps contain.

Statement



Use Statement steps, shown at left, to execute expressions. For example, you can use a Statement step to increment the value of a local variable in a sequence.

By default, Statement steps do not pass or fail. If the step cannot evaluate the expression or if the expression sets `Step.Result.Error.Occurred` to `True`, TestStand sets the step status to `Error`. Otherwise, TestStand sets the step status to `Done`.

Refer to the *NI TestStand Help* for more information about the Expression edit tab.

The Statement step type does not define any additional step properties other than the custom properties all steps contain.

Label



Use a Label step, shown at left, as the target for a Goto step, which you can use to rearrange or delete other steps in a sequence without having to change the specification of targets in Goto steps.

Label steps do not pass or fail and by default do not record results. After a Label step executes, the TestStand Engine sets the step status to `Done` or

Error. You can edit a Label step to specify a description that appears next to the Label step name in the sequence editor.

Refer to the *NI TestStand Help* for more information about the Label step edit tab.

In addition to the common custom properties, the Label step type defines the following step property:

- **Step.Description**—Specifies a string that appears next to the step name in the sequence editor.

Message Popup



Use Message Popup steps, shown at left, to display messages to the user and to receive response strings from the user. For example, you can use a Message Popup step to warn the user when a calibration routine fails.

By default, Message Popup steps do not pass or fail. After a step executes, TestStand sets the step status to **Done** or **Error**.

Refer to the *NI TestStand Help* for more information about the Message Popup step edit tab.

In addition to the common custom properties, the Message Popup step type defines the following step properties:

- **Step.Result.ButtonHit**—Contains the one-based index of the button you select.
- **Step.Result.Response**—Contains the response text the user entered.
- **Step.TitleExpr**—Contains the expression for the string that appears as the title of the message popup dialog box.
- **Step.MessageExpr**—Contains the expression for the string that appears as the text message in the message popup dialog box.
- **Step.MessageFontData**—Specifies the font for the text message in the message popup dialog box.
- **Step.Button1Label**, **Button2Label**, **Button3Label**, **Button4Label**, **Button5Label**, and **Button6Label**—Specify the expression for the label text for each button.
- **Step.ButtonFontData**—Specifies the font for the label text for buttons in the message popup dialog box.
- **Step.ShowResponse**—Enables the response text box control in the message popup dialog box.

- **Step.NumberLines**—Specifies the number of visible text lines in the response text box.
- **Step.MaxResponseLength**—Specifies the maximum number of characters the user can enter in the response text box control.
- **Step.RespFontData**—Specifies the font for the response text box control in the message popup dialog box.
- **Step.DefaultResponseExpr**—Contains the initial text string the step displays in the response text box control.
- **Step.FileData**—Specifies to display a graphic or Web page in the message popup dialog box.
- **Step.ActiveCtrl**—Identifies one of the six buttons or the input string as the active control.
- **Step.DefaultButton**—Specifies which button, if any, uses **<Enter>** as a shortcut key.
- **Step.CancelButton**—Specifies which button, if any, uses **<Esc>** as a shortcut key.
- **Step.TimerButton**—Specifies the index of the button that activates automatically after a timeout elapses. A value of zero indicates no timeout occurs.
- **Step.TimeToWait**—Specifies the number of seconds before the button **Step.TimerButton** specifies activates.
- **Step.Position.Top** and **Step.Position.Left**—Specify the location of the message popup dialog box when **CenterDialog** is **False**.
- **Step.CenterDialog**—Specifies if the message popup dialog box appears in the center of the screen.
- **Step.Modal**—Specifies if the message popup dialog box is modal to the TestStand application.
- **Step.Floating**—Specifies if the message popup dialog box stays on top of the TestStand application.
- **Step.CtrlArrangement**—Specifies the order for the controls in the message popup dialog box.
- **Step.ButtonLocation**—Specifies to display the buttons on the bottom or side of the message popup dialog box.
- **Step.ButtonAlignment**—Specifies to align the buttons in the center, left, right, top, or bottom of the message popup dialog box.
- **Step.ResizeDialog**—Specifies if the message popup dialog box is resizable.

Call Executable



Use Call Executable steps, shown at left, to launch an application or run a system command. For example, you can use a Call Executable step to call a system command to copy files to a network drive.

The final status of a Call Executable step depends on if the step waits for the executable to exit. If the step does not wait for the executable to exit, the step type always sets the step status to `Done`. If a timeout occurs while the step is waiting for the executable to exit, the step type sets the status to `Error`. If the step waits for the executable to exit and a timeout does not occur, the step type sets the step status to `Done`, `Passed`, or `Failed`, depending on the status action you specify in the Exit Code Status Action ring control on the Call Executable edit tab for the step. If you set the Exit Code Status Action ring control to `No Action`, the step type always sets the step status to `Done`. Otherwise, you can choose to set the step status to `Failed` based on if the exit code is less than zero, greater than zero, equal to zero, or not equal to zero.

Refer to the *NI TestStand Help* for more information about the Call Executable step edit tab.

In addition to the common custom properties, the Call Executable step type defines the following step properties:

- **Step.Result.ExitCode**—Contains the exit code the executable returns.
- **Step.Executable**—Specifies the pathname of the executable to launch.
- **Step.Arguments**—Specifies the expression for the argument string the step passes to the executable.
- **Step.WaitCondition**—Specifies if the step waits for the executable to exit before completing.
- **Step.TimeToWait**—Specifies the number of seconds to wait for the executable to exit.
- **Step.InitialWindowState**—Specifies if the executable is initially active, not active, hidden, normal, minimized, or maximized.
- **Step.TerminateOnAbort**—Specifies to terminate the executable process when the execution terminates or aborts.
- **Step.ProcessHandle**—Contains the Windows process handle for the executable.
- **Step.ExitCodeStatusAction**—Specifies to set the step status using the exit code the executable returns.

- **Step.RemoteSettings**—Contains the following settings for calling the executable on a remote computer:
 - **Enabled**—Specifies to call the executable on a remote computer.
 - **Host**—Specifies the computer name or IP address of the remote computer.
 - **HostByExpr**—Specifies if you can use an expression in the Remote Host field.
 - **Port**—Specifies the port number the remote host server application uses.
 - **Password**—Specifies the password configured on the remote host server application.
 - **PasswordByExpr**—Specifies if you can use an expression in the Password field.

Property Loader



Use the Property Loader step type, shown at left, to dynamically load property and variable values from a text file, a Microsoft Excel file, or a database management system (DBMS) at run time. Refer to Appendix C, [Database Step Types](#), for more information about the Property Loader step. Refer to the *NI TestStand Help* for more information about the Edit Property Loader dialog box.

FTP Files



Use an FTP Files step, shown at left, to transfer files between the local system and an FTP server.

Refer to the *NI TestStand Help* for more information about the FTP Files step edit tab.

In addition to the common custom properties, the FTP Files step type defines the following step properties:

- **Step.RemoteHost**—Specifies the computer name or IP address of the remote computer.
- **Step.RemoteHostByExpr**—Specifies if you can use an expression in the Hostname field.
- **Step.FTPUsername**—Specifies the login name to use when connecting to the server.

- **Step.FTPPassword**—Specifies the password to use when connecting to the server.
- **Step.FilesToFTP**—Specifies the local and remote file paths, the direction to transfer the file, and to overwrite a file if it exists.

Additional Results



Use an Additional Results step, shown at left, to specify values to add to the result list of a step. Optionally include the values in a report or log the values to a database. An additional result is a value TestStand adds to the result list of a step when the step executes. An additional result can be a module parameter or a custom additional result in which you specify the name and value of the result.

Refer to the *NI TestStand Help* for more information about the Additional Results edit tab.

Synchronization Step Types

Refer to Appendix B, *Synchronization Step Types*, for more information about the Synchronization steps. Refer to the *NI TestStand Help* for more information about the Synchronization step edit tabs.

Database Step Types

Refer to Appendix C, *Database Step Types*, for more information about the Database steps. Refer to the *NI TestStand Help* for more information about the Database step edit dialog boxes.

IVI Step Types

Refer to Appendix D, *IVI Step Types*, for more information about the IVI steps. Refer to the *NI TestStand Help* for more information about the IVI step edit dialog boxes.

LabVIEW Utility Step Types

Refer to Appendix E, *LabVIEW Utility Step Types*, for more information about the LabVIEW Utility steps. Refer to the *NI TestStand Help* for more information about the LabVIEW Utility step edit tabs.

Module Adapters

The TestStand Engine uses module adapters to invoke code modules. TestStand sequences call. Module adapters load and call code modules, pass parameters to code modules, and return values and status from code modules. The module adapters support the following types of code modules:

- LabVIEW VIs
- LabWindows/CVI functions in source files, object files, or library modules you create in LabWindows/CVI or other compilers
- C/C++ functions in DLLs
- .NET assemblies
- ActiveX Automation servers
- HTBasic subroutines

When you edit a step that uses a module adapter, TestStand displays the Module tab on the Step Settings pane, where you specify the code module for the step and specify parameters to pass when you invoke the code module. TestStand stores the name and location of the code module, the parameter list, and any additional options as built-in properties of the step.

ADE-specific adapters can open the ADE, create source code for a new code module in the ADE, and display the source for an existing code module in the ADE. The adapters support stepping into the source code in the ADE while you execute the step from the TestStand Sequence Editor or user interfaces.

Configuring Adapters

Select **Configure»Adapters** from the sequence editor menu to configure the module adapters. Refer to the *NI TestStand Help* for more information about configuring adapters.

Source Code Templates

With the LabVIEW, LabWindows/CVI, C/C++ DLL, .NET, and HTBasic Adapters, you can use a code template to generate a source code shell for a code module. The code template files are different for each step type and each module adapter. A step type can define multiple code templates for an adapter/step combination.

TestStand includes default code templates for each built-in step type. You can also create additional code templates for built-in step types when you create a new step type. Refer to the [Code Templates Tab](#) section of Chapter 13, [Custom Step Types](#), for more information about creating code templates for step types.

Search Paths

TestStand includes a list of search directories module adapters use to resolve relative paths of code modules for steps and substeps in step types and to locate code modules when executing steps. TestStand also uses the search directories to resolve relative pathnames for files and directories when calling the TestStand API `Engine.FindFile` and `Engine.FindPath` methods. Refer to the *NI TestStand Help* for more information about the TestStand API. Refer to the [Substeps Tab](#) section of Chapter 13, [Custom Step Types](#), for more information about substeps.

Select **Configure»Search Directories** in the TestStand Sequence Editor to launch the Edit Search Directories dialog box, in which you can view and edit the default list of search paths. The list of default directories includes specific TestStand directories and Windows system directories, and you can add custom directories to the list. Use relative paths if possible when you add directories. The paths that appear first in the list take precedence over the paths that appear later in the list. You can exclude directories, reorder directories, specify to recursively search directories, and specify file extension restrictions for directories. Refer to the *NI TestStand Help* for more information about the Edit Search Directories dialog box.

TestStand includes the following directories by default, in order of precedence:

- Current sequence file directory
- Current workspace directory
- Application directory (disabled by default)
- <TestStand> directory

- <TestStand>\bin directory
- Initial working directory (disabled by default)
- Windows system directory
- Windows directory
- PATH environment variable (disabled by default)
- <TestStand Public> directory
- <TestStand Public>\Components directory
- <TestStand>\Components directory

If you list a directory and a subdirectory within that directory, TestStand performs a double search. You might want to use a double search only if both directories contain a file with the same name but different content. In most cases, include only the higher level directory.

Refer to the [TestStand Directory Structure](#) section of Chapter 8, [Customizing and Configuring TestStand](#), for more information about TestStand directories.

Configuring Search Paths for Deployment

When you want to configure search directories for deploying a TestStand system, you can manually add additional search paths to the list of default search paths on the target computer. The TestStand Deployment Utility does not copy additional search paths because the new directories might not exist on the target computer.

National Instruments recommends installing the <TestStand Application Data>\Cfg\TestExec.ini file with the deployment for that system because TestStand stores the search directories in TestExec.ini. However, a limitation in the TestStand installer prevents a non-versioned file from replacing an existing file on a system if the date of the file you replace is more recent than the one you install. Because the TestStand Engine updates TestExec.ini when TestStand runs the file, you might encounter this limitation when you try to install a deployment over a previously installed deployment. You can work around this limitation by manually removing the file before you install the deployed TestStand system or by adding a command to a script you execute before you invoke the installer. Refer to Chapter 14, [Deploying TestStand Systems](#), for more information about building a TestStand system for deployment.

LabVIEW Adapter

Use the LabVIEW Adapter to call LabVIEW VIs with a variety of connector panes. Use the LabVIEW Module tab on the Step Settings pane to configure calls to LabVIEW VIs. Refer to the *NI TestStand Help* for more information about the LabVIEW Module tab and passing parameters between TestStand and VIs. Refer to the *Using LabVIEW with TestStand* manual for more information about using the LabVIEW Adapter, supported data types, and tutorials that use the adapter.

LabWindows/CVI Adapter

Use the LabWindows/CVI Adapter to call C functions with a variety of parameter types. The function can exist in an object file, library file, or DLL. The function can also exist in a source file located in the project you are currently using in the LabWindows/CVI development environment. Use the LabWindows/CVI Module tab on the Step Settings pane to configure calls to LabWindows/CVI code modules.

Refer to the [Debugging DLLs](#) section of this chapter for more information about debugging DLLs built with LabWindows/CVI. Refer to the *NI TestStand Help* for more information about the LabWindows/CVI Module tab and passing parameters between TestStand and code modules. Refer to the *Using LabWindows/CVI with TestStand* manual for more information about using the LabWindows/CVI Adapter, supported data types, and tutorials that use the adapter.

C/C++ DLL Adapter

Use the C/C++ DLL Adapter to call C functions and C++ methods in a DLL with a variety of parameter types. You can call global static methods or static class methods in C++ DLLs. You can create DLL code modules with Microsoft Visual Studio, LabWindows/CVI, or any other ADE that creates a C/C++ DLL you can call.

Use the C/C++ DLL Module tab on the Step Settings pane to specify a C/C++ DLL Adapter module call, to specify the source code associated with the module call, and to create and edit C/C++ code modules directly from TestStand.

If you use Visual Studio, you must have National Instruments Measurement Studio 8.0.1 (or later) Enterprise Edition and Visual Studio 2005 or later installed.

For DLLs built with LabWindows/CVI, you must use the LabWindows/CVI Adapter to create and edit code modules directly from TestStand. The LabWindows/CVI Adapter provides full integration with the LabWindows/CVI ADE for debugging.

You can also use a text editor to create and edit code directly from TestStand.

Refer to the *NI TestStand Help* for more information about using the C/C++ DLL Adapter, the C/C++ DLL Module tab, and passing parameters between TestStand and code modules.

Using DLLs

You can call LabVIEW, MFC, and subordinate DLLs from TestStand, which can display parameter information on the Module Tab on the Step Settings pane if the DLL contains export information.

Using ActiveX Controls in LabVIEW DLLs

LabVIEW shared libraries (DLLs) that use ActiveX controls must load in a thread initialized as single-threaded apartment (STA) for the controls to function correctly. If the TestStand step that calls the DLL preloads the DLL, TestStand ensures that the DLL loads in an STA thread. However, if you dynamically load a step that calls the DLL, you must ensure that the loading sequence executes in an STA thread.

Use the Run Sequence in a New Thread option or the Run Sequence in a New Execution option located in the Multithreading and Remote Execution section in the Edit Sequence Call dialog box to select an STA thread. Click the Settings button in the Edit Sequence Call dialog box to launch the Thread Settings dialog box, which contains the STA thread options.

Using MFC in DLLs

The Microsoft Foundation Class (MFC) Library places several requirements on DLLs that use the DLL version of the MFC run-time library. If you call a DLL that includes MFC functions, verify that the DLL meets these requirements. Also, if the DLL uses resources such as dialog boxes, verify that the `AFX_MANAGE_STATE` macro appears at the beginning

of the function body of each function you call. Refer to the MFC documentation for more information about calling DLLs.

Loading Subordinate DLLs

TestStand directly loads and runs the DLLs you specify on the C/C++ DLL Module tab of the C/C++ DLL 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 C/C++ DLL Adapter attempts to load subordinate DLLs using the following search directory precedence:

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

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

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



Note National Instruments does not recommend placing subordinate DLLs in the directory that contains the application that loads the adapter because TestStand might not support loading DLLs from this location in future versions.

Reading Parameter Information

If a DLL contains export information or if a DLL file contains a type library, the LabWindows/CVI and C/C++ DLL Adapters automatically populate the Function control on the Module tab of the step 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 displays it in the Parameter Table control on the Module tab. If the adapter cannot determine parameter information, you must enter the parameter information manually.

Refer to Appendix B, *Adding Type Libraries to LabWindows/CVI DLLs*, of the *Using LabWindows/CVI with TestStand* manual for more information about using a function panel file to generate a type library to include in a DLL.

Debugging DLLs

To debug a DLL TestStand calls, first create the DLL with debugging enabled in the ADE. Then, launch the sequence editor or user interface executable from the ADE or attach to the sequence editor or user interface process from the ADE, if supported.



Note Save sequence files and workspaces before you stop debugging and terminate the TestStand process because most ADEs terminate the process without prompting you to save modified files in TestStand.

For LabWindows/CVI, select **Run»Select External Process** in the Project window to identify the executable for the sequence editor or user interface and select **Run»Debug <executable name>** to start debugging the executable. For Visual Studio, you must enable native code debugging.

If you suspend a sequence on a step that calls a debuggable DLL, click the **Step Into** button in TestStand to suspend at the first statement in the DLL function within LabWindows/CVI or Visual Studio 2005.



Note You cannot debug DLLs the C/C++ Adapter calls using Visual Studio .NET 2003 or using Visual Studio 2005 when the process loads Microsoft .NET Framework 1.1.

To step into a code module with LabWindows/CVI, you must configure the step to use the LabWindows/CVI Adapter. You can step into a code module when you configure the LabWindows/CVI Adapter to execute steps in-process or in an external instance of LabWindows/CVI.

To step into a DLL with Visual Studio 2005, you must configure the step to use the C/C++ DLL Adapter and you must have Measurement Studio 8.0.1 or later Enterprise Edition installed. If you attempt to step into a DLL while Visual Studio is not attached to the TestStand process, TestStand launches Visual Studio, which automatically attaches to the TestStand process using native debugging.

Table 5-1 lists the options for stepping out of a LabWindows/CVI or Visual Studio DLL function.

Table 5-1. Options for Stepping Out of DLL Functions

ADE Command for Stepping Out	Result in TestStand
Finish Function or Step Out	Function executes. When you use this command on the last function in the call stack, TestStand suspends execution on the next step in the sequence.
Step Into or Step Over	When you use this command on the last executable statement of the function, TestStand suspends execution on the next step in the sequence.
Continue	TestStand does not suspend execution when the function call returns.



Note If the Step Over command executes on an END step in a Pre-Step callback, TestStand attempts to step into the code module.

Refer to the LabWindows/CVI and Visual Studio documentation for more information about debugging DLLs in an external process.

Debugging LabVIEW 8.0 and Later Shared Libraries (DLLs)

With LabVIEW 8.0 and later, you can enable debugging in shared libraries you build with the Application Builder. Using the LabVIEW development environment, you can configure the shared library to debug in and connect to the TestStand application process. Refer to the *LabVIEW Help* for more information about debugging applications and shared libraries.

Debugging LabVIEW 7.1.1 Shared Libraries (DLLs)

You must use a TestStand User Interface built with LabVIEW to debug any VIs you include in a LabVIEW 7.1.1 shared library. Before you open and run the user interface in the LabVIEW development environment, open the VI that represents the DLL function in the shared library you want to debug.

and place a breakpoint on the block diagram. Use the TestStand User Interface to load and execute the sequence file that calls the LabVIEW shared library. When LabVIEW loads the shared library the step calls, LabVIEW uses the VI in memory instead of the VI in the DLL. When the step calls the DLL function, LabVIEW suspends at the breakpoint you set in the VI.

.NET Adapter

Use the .NET Adapter to call .NET assemblies written in any .NET-compliant language, such as C# or Visual Basic .NET. You must have the .NET Framework 2.0 or later installed to use the .NET Adapter.

Use the .NET Module tab on the Step Settings pane to configure calls to .NET assemblies.

With the .NET Adapter, you can create instances of classes and structs, call methods, and access properties or fields on classes and structs. With an instance of a class you previously created and stored in an object reference variable or created in the calling step, you can call or access all non-static public members. You do not need to use an instance to call or access static public members. When you call a struct, you can store the definition in a variable of a data type mapped to the struct members or initialized in the calling step.

The .NET Adapter does not support creating or calling methods on generic classes.

You can create and edit .NET code modules in Visual Studio directly from TestStand if you install National Instruments Measurement Studio 8.0.1 (or later) Enterprise Edition.

Refer to the *NI TestStand Help* for more information about using the .NET Adapter, the .NET Module tab, and passing parameters between TestStand and code modules.

Debugging .NET Assemblies

To debug a .NET assembly, first create the assembly with debugging enabled in the ADE. Then, launch the sequence editor or user interface from Visual Studio or attach to the sequence editor or user interface process from Visual Studio.



Note Save sequence files and workspaces before you stop debugging and terminate the TestStand process because Visual Studio might terminate the process without prompting you to save modified files in TestStand.

If you use Visual Studio 2005 or later, you have Measurement Studio 8.0.1 (or later) Enterprise Edition installed, and you suspend a sequence on a step that calls a debuggable assembly, click the **Step Into** button in TestStand to suspend Visual Studio at the first statement in the assembly method or property.

If you attempt to step into an assembly while Visual Studio is not attached to the TestStand process, TestStand launches Visual Studio, which automatically attaches to the TestStand process using managed debugging. When you debug managed code in a TestStand process with Visual Studio, TestStand does not unload assemblies when you select **File»Unload All Modules**.



Note You cannot debug managed code the .NET Adapter calls using Visual Studio .NET 2003 or using Visual Studio 2005 when the process loads the .NET Framework 1.1. You cannot attach to a TestStand process when the process loads the .NET Framework 1.1 and you are debugging with Visual Studio 2005 or when the process loads the .NET Framework 2.0 and you are debugging with Visual Studio .NET 2003.

Table 5-2 lists the options for stepping out of a Visual Studio assembly.

Table 5-2. Options for Stepping Out of Assemblies in Visual Studio

Visual Studio Command for Stepping Out	Result in TestStand
Step Out	Function executes. When you use this command on the last function in the call stack, TestStand suspends execution on the next step in the sequence.
Step Into or Step Over	When you use this command on the last executable statement of the function, TestStand suspends execution on the next step in the sequence.
Continue	TestStand does not suspend execution when the function call returns.

Refer to the Visual Studio documentation for more information about debugging managed code in an external process.



Note If you use LabWindows/CVI to debug a DLL in the TestStand process, you cannot debug a .NET assembly at the same time. If you use Visual Studio to debug an assembly in TestStand and you want to use LabWindows/CVI to debug code modules at the same time, you must configure the LabWindows/CVI Adapter to execute the steps in an external instance of LabWindows/CVI.

Using the .NET Framework

An application can load only one version of the .NET runtime into memory. For unmanaged applications, such as the LabVIEW user interface, the .NET Adapter uses the latest version of the .NET runtime. For managed applications, such as the TestStand Sequence Editor and the C# and Visual Basic .NET user interfaces, the .NET Adapter uses the runtime specified when the application was created.

You can call .NET 1.1 assemblies in TestStand with the .NET 2.0 runtime loaded in memory, but you cannot call .NET 2.0 assemblies in TestStand with the .NET 1.1 runtime loaded in memory. Also, you cannot use Visual Studio .NET 2003 to debug the Common Language Runtime with the .NET 2.0 runtime loaded in memory.

You can force an application to use a specific version of the .NET Framework by creating a configuration file in the same directory as the executable. For example, to force an unmanaged user interface to use the .NET Framework 1.1, create the following `testexec.exe.config` file:

```
<?xml version="1.0"?>
  <configuration>
    <runtime>
      <assemblyBinding
        xmlns="urn:schemas-microsoft-com:asm.v1">
      </assemblyBinding>
    </runtime>
    <startup>
      <supportedRuntime version="v1.1.4322" />
    </startup>
  </configuration>
```

Accessing the TestStand API in Visual Studio .NET 2003 and Visual Studio 2005

TestStand installs .NET interop assemblies for the TestStand API in the `<TestStand>\API\DotNet\Assemblies` directory and adds references to the assemblies to the Global Assembly Cache (GAC). The interop assemblies support the current and earlier versions of the TestStand API. The TestStand 4.0 and later assemblies require the .NET 2.0 runtime, and the TestStand 3.5 and earlier assemblies require the .NET 1.1 or later runtime.

To add a reference to the TestStand 4.0 and later API assembly in Visual Studio 2005, select the project in the Solution Explorer. Select **Project»Add Reference** to launch the Add Reference dialog box. Click the **.NET** tab and select the TestStand `<APIName> Interop Assembly` component from the list. Click **OK** to close the Add Reference dialog box.

To add a reference to the TestStand API assembly in Visual Studio .NET 2003, you must use a .NET 1.1-compatible assembly, such as TestStand 3.5 or earlier. Select the project in the Solution Explorer. Select **Project»Add Reference** to launch the Add Reference dialog box. Click the **.NET** tab and click the **File Browse** button to launch the Select Components dialog box. Navigate to the `<TestStand>\API\DotNet\Assemblies\PreviousVersion\<VersionNumber>` directory. Select `NationalInstruments.TestStand.Interop.<APIName>.dll` and click **Open**. Click **OK** to close the Add Reference dialog box.

ActiveX/COM Adapter

Use the ActiveX/COM Adapter to create objects, call methods, and access properties of ActiveX/COM objects. When you create an object, you can assign the object reference to a variable or property for later use in other ActiveX/COM Adapter steps. When you call methods and access properties, you can specify an expression for each input and output parameter.

Use the ActiveX/COM Module tab on the Step Settings pane to configure calls to ActiveX/COM servers.

Refer to the *NI TestStand Help* for more information about the ActiveX/COM Module tab and configuring calls to ActiveX/COM servers.

Debugging ActiveX Automation Servers

TestStand does not step into ActiveX/COM servers. To debug an out-of-process executable server, launch the server in the ADE in which it was created and independently launch the sequence editor or user interface. If you want to debug an in-process DLL server, launch the sequence editor or user interface from the ADE, or attach to the sequence editor or user interface process from the ADE, if supported.

When you work in Microsoft Visual Basic, place breakpoints in the automation server source code and select **Run»Start with Full Compile**. In TestStand, run the sequence that calls into the automation server to cause the execution to automatically suspend at the breakpoint you set in Visual Basic. Refer to the ADE documentation for more information about debugging ActiveX automation servers.



Note When TestStand requests that the Windows operating system unload a DLL server, the operating system ignores the request because TestStand is still using the DLL server. The operating system keeps the DLL server in memory, which prevents the development environment from rebuilding the DLL. You must exit the TestStand application to release the DLL server.

Registering and Unregistering ActiveX/COM Servers

To register an ActiveX/COM server DLL, call the Windows executable `regsvr32.exe` and use the DLL pathname as the command-line argument. To unregister the DLL server, call `regsvr32.exe` and use `/u` and the DLL pathname as the command-line argument.

To register an ActiveX/COM server executable, run the server executable with the `/RegServer` command-line argument. To unregister an executable server, run the executable with the `/UnregServer` command-line argument.

Visual Basic does not automatically register a server when you build the server DLL or executable. You must manually register the server. Visual Basic temporarily registers a server when you run the server project inside the Visual Basic ADE. When you complete the debugging session, Visual Basic unregisters the server.

Server Compatibility Options for Visual Basic

If you develop an automation server in an ADE that does not give you direct control over IDs, you must ensure that the ActiveX/COM Adapter can find the server identifiers or the names you define in the TestStand step.

When you rebuild an ActiveX/COM server in Visual Basic, you can select a compatibility option. Depending on the level of compatibility and the changes you make to a project, Visual Basic compiles an appropriate new server, which can contain new identifiers.

Select **Project»Project Properties** in Visual Basic to specify the level of compatibility. On the Components tab of the Project Properties dialog box, select one of the following options in the Version Compatibility section:

- **No compatibility**—Maintains no compatibility between the new server and a previously compiled server. Visual Basic generates new unique identifiers for the server, which prevents any previously compiled client application that uses early binding from working properly with the server.

Because Visual Basic changes the IDs it uses to uniquely identify the type information of the server, TestStand cannot properly update an ActiveX/COM Adapter step, regardless if you configure the ActiveX/COM Adapter for early or late binding.



Note National Instruments does not recommend using the No compatibility setting with TestStand projects.

- **Project compatibility**—Maintains the current ID assignments Visual Basic uses to uniquely identify the type information for the server. Use this option when you are developing multiple projects with Visual Basic. Using this setting does not ensure compatibility with client applications not compiled in Visual Basic or with compiled client applications that use early binding.

When you use this option to rebuild a server, TestStand can use the type information to determine the IDs associated with the names stored in the step.

Use the Project compatibility option to rebuild the server in Visual Basic while you develop and debug sequences. Configure the ActiveX/COM Adapter to use late binding.

- **Binary compatibility**—Maintains the current ID assignments Visual Basic uses to uniquely identify objects and methods. Visual Basic attempts to maintain compatibility with compiled client applications that use early binding. If you remove a member from the server, Visual Basic can no longer maintain binary compatibility.

When you use this option to rebuild a server, TestStand can use the IDs stored in the step without accessing the type information at run time.

Use the Binary compatibility option to rebuild the server in Visual Basic when the interface for the server is completely developed and debugged. Select **Tools»Update Automation Identifiers** in the TestStand Sequence Editor to assign the new server identifiers to the steps. Enable the ActiveX/COM Adapter to use early binding.

Refer to the Visual Basic documentation and to the following Internet document for more information about creating and debugging Visual Basic ActiveX/COM servers:

Ivo Salmre, “Building, Versioning, and Maintaining Visual Basic Components,” *Microsoft Developer Network*, Microsoft Corporation, February 1998.

HTBasic Adapter

Use the HTBasic Adapter to call HTBasic subroutines without passing parameters directly to a subroutine. TestStand provides a library of CSUB routines that use the TestStand API to access TestStand variables and properties from an HTBasic subroutine. Refer to the *NI TestStand Help* for more information about HTBasic subroutines.



Note HTBasic currently does not support Windows Vista. However, if you installed the HTBasic 9.0 development environment under Windows Vista, you can still perform the Edit Subroutine and Create Subroutine functions on the HTBasic Module tab in the TestStand Sequence Editor when you use a step configured to use the HTBasic Adapter. However, HTBasic code modules might not run correctly.

Use the HTBasic Module tab on the Step Settings pane to specify the subroutine file path, subroutine name, and other options. Refer to the *NI TestStand Help* for more information about the HTBasic Module tab.

Debugging HTBasic Subroutines

To debug an HTBasic subroutine while executing the subroutine from TestStand, you must configure the adapter to use the HTBasic development environment as the HTBasic server.

If you suspend a sequence on a step that calls an HTBasic subroutine, click the **Step Into** button in TestStand to display the HTBasic server window and pause at the call of the subroutine. Press <Alt-F1> to single-step through the subroutine. When you finish debugging a particular subroutine,

click **Continue** to resume execution and return control to TestStand. After you step out of the subroutine, TestStand suspends execution on the next step in the sequence.

Refer to the HTBasic documentation for more information about debugging HTBasic programs.

Sequence Adapter

Use the Sequence Adapter to pass parameters when you make a call to a subsequence. You can call a subsequence in the current sequence file or in another sequence file, and you can make recursive sequence calls. For subsequence parameters, you can specify a literal value, pass a variable or property by reference or by value, or use the default value the subsequence defines for the parameter.

Use the Sequence Module tab on the Step Settings pane to specify a Sequence Adapter module call.

You can use the Sequence Adapter from any step type that can use module adapters, such as the Pass/Fail Test or the Numeric Limit Test. Using the Sequence Adapter this way is similar to using the built-in Sequence Call step type, except that the Sequence Call step sets the step status to `Passed` instead of `Done` if no failure or error occurs.

After the Sequence Call step executes, the Sequence Adapter can set the step status. If no run-time error occurs, the adapter does not set the step status, which is `Done` or `Passed`, depending on the type of step. If the sequence the step calls fails, the adapter sets the step status to `Failed`. If a run-time error occurs in the sequence, the adapter sets the step status to `Error` and sets the `Result.Error.Occurred` property to `True`. The adapter also sets the `Result.Error.Code` and `Result.Error.Msg` properties to the values of these same properties in the subsequence step that generated the run-time error.

Use the Variables pane in the Sequence File window to define parameters for a sequence, including the parameter name, its TestStand data type, its default value, and whether to pass the argument by value or by reference. Refer to the *NI TestStand Help* for more information about the Sequence Module tab and sequence file parameters.

Remote Sequence Execution

When you specify a sequence file pathname on the Sequence Module tab and specify Use Remote Computer in the Execution Options section, TestStand locates the sequence file according to the type of path, as described in Table 5-3.

Table 5-3. Path Resolution of Sequence Pathnames for Remotely Executed Steps

Type of Path	Where Found When You Edit	Where Found When You Execute	Example
Relative	In the TestStand search paths you configure on the client (local) computer	In the TestStand search paths you configure on the server (remote) computer	Transmit.seq
Absolute	On the client (local) computer	On the server (remote) computer	C:\Projects\Transmit.seq
Network	On the computer the network path specifies	On the computer the network path specifies	\\Remote\Projects\Transmit.seq

When you edit a step in a sequence file on a client computer and you specify an absolute or relative path for the sequence file the step calls, TestStand resolves the path for the sequence file on the client computer. When you execute the step on the client computer, TestStand resolves the path for the sequence file on the server computer.

You can manage remote sequence files for remote execution in the following ways:

- Add a common pathname to the search paths for the client and the server computers so that each resolves to the same relative pathname. Refer to the [Search Paths](#) section of this chapter for more information about TestStand search directories.
- Duplicate the files on the client and the server computers so that the file you edit on the client computer is identical to the file the server computer executes.
- Use absolute paths that specify a mapped network drive or full network path so that the file the client computer edits and the file the server computer executes are the same sequence file.

When you execute a remote sequence, you cannot single-step or set breakpoints in the remote sequence. If you enable tracing, TestStand updates the status bar with tracing information for the remote sequence.

When a remote sequence executes on a server, the sequence context and call stack include only the sequences that run on the remote computer. If you want to access properties from the client sequence context, you must pass the `PropertyObject` objects or their values as parameters to the remote sequence. You can use the TestStand API to access properties within a property object.

If you want to use the TestStand API on the server computer to access objects on a client computer that runs Windows XP Service Pack 2 (SP2) or Windows Vista, you must configure the Distributed COM (DCOM) access permissions on the client computer. Complete the following steps to configure DCOM access permissions on the client computer. You must restart TestStand for these changes to take effect.

1. Log in as a user with administrator privileges.
2. **(Windows XP)** Navigate to **Administrative Tools** on the Windows Control Panel and select **Component Services** or run `dcomcnfg` from the command line to launch the Component Services window.
(Windows Vista) Run `dcomcnfg` from the command line or open `<Windows>\system32\comexp.msc` to launch the Component Services window.
3. In the left pane of the Component Services window, select **Component Services»Computers»My Computer**.
4. Right-click **My Computer** and select **Properties** to launch the My Computer Properties dialog box.
5. Click the **COM Security** tab of the My Computer Properties dialog box.
 - a. Click the **Edit Limits** button in the Access Permissions section to launch the Access Permissions dialog box.
 - b. Select **ANONYMOUS LOGON** in the user name list and enable **Remote Access** in the Permissions Section.
 - c. Click **OK** to close the Access Permissions dialog box.

You must properly configure a remote computer and the TestStand server application on the remote computer if you want to invoke a sequence on the remote computer from TestStand on a client computer. You must enable the TestStand server on the remote computer to accept remote execution requests. You must also configure Windows system security to allow users

to access and launch the TestStand server remotely. For Windows XP SP2, you must also configure the Windows Firewall on the remote computer.

Setting up TestStand as a Server for Remote Sequence Execution

Enable the **Allow Sequence Calls from Remote Machines to Run on this Machine** option located on the Remote Execution tab of the Station Options dialog box to allow the TestStand server to accept remote execution requests from a client computer.

A TestStand server is active while the TestStand application <TestStand>\bin\REngine.exe runs on a remote computer. Each TestStand client communicates with a dedicated version of the TestStand server, which launches automatically when a TestStand client uses the server.

Enable the **Show the System Tray Icon While the TestStand Remote System is Active on this Machine** option in the Station Options dialog box on the remote computer to make the TestStand icon visible in the remote computer system tray for each instance of the remote engine application. The tooltip for the icon indicates which computer is connected to the remote engine. Right-click the TestStand icon to display when the engine was created or to force the remote engine application to close.

TestStand automatically registers the server during installation. To manually register or unregister the server, invoke the executable with the /RegServer or /UnregServer command-line arguments.

Setting Windows System Security

To minimize the configuration of security permissions, enable the **Allow All Users Access From Remote Machines** option in the Station Options dialog box.

(Windows Vista) Windows Vista launches a User Account Control elevation prompt for you to manually resolve when you enable this option.

When you enable the Allow All Users Access From Remote Machines option, TestStand configures the security permissions for you by adding the name Everyone for Windows 2000 Service Pack 4 (SP4) and the name ANONYMOUS LOGON for Windows XP SP2 and Windows Vista to the list of users who have permission to access and launch the TestStand remote server. When you disable this option, TestStand removes the names from the list.

Windows XP Service Pack 2 and Windows Vista

Complete the following steps to configure the security permissions for the server on a remote computer.

1. Log in as a user with administrator privileges.
2. **(Windows XP)** Navigate to **Administrative Tools** on the Windows Control Panel and select **Component Services** or run `dcomcnfg` from the command line to launch the Component Services window.
(Windows Vista) Run `dcomcnfg` from the command line or open `<Windows>\system32\comexp.msc` to launch the Component Services window.
3. In the left pane of the Component Services window, select **Component Services»Computers»My Computer**.
4. Right-click **My Computer** and select **Properties** to launch the My Computer Properties dialog box.
5. On the **Default Properties** tab of the My Computer Properties dialog box, enable the **Enable Distributed COM on this computer** option.



Note You must restart the computer for changes to the value of the **Enable Distributed COM on this computer** option to take effect.

6. Click the **COM Security** tab of the My Computer Properties dialog box.
 - a. Click the **Edit Limits** button in the Access Permissions section to launch the Access Permission dialog box. Click **Add** to add the users you want to give remote access to. Click **OK** to close the Select Users, Computer, Groups dialog box. Select the user you added and enable **Remote Access** in the Permissions section. Click **OK** to close the Access Permission dialog box.
 - b. Click the **Edit Limits** button in the Launch and Activation Permissions section to launch the Launch Permission dialog box. Click **Add** to add the users you want to give remote access to. Click **OK** to close the Select Users, Computer, Groups dialog box. Select the user you added and enable **Remote Launch** and **Remote Activation** in the Permissions section. Click **OK** to close the Launch Permission dialog box.



Note You can grant access permission to the remote computer to everyone but grant launch permissions only to appropriate users because launch permissions allow access to the TestStand server on the remote computer.

7. Click **OK** to close the My Computer Properties dialog box.
8. In the left pane of the Component Services window, select **My Computer»DCOM Config** to display a list of applications on the right pane.
9. Right-click **NI TestStand Remote Engine** and select **Properties** from the context menu to launch the NI TestStand Remote Engine Properties dialog box.
10. On the **Identity** tab of the NI TestStand Remote Engine Properties dialog box, select the **The interactive user** option. Click **OK** to close the dialog box.

Windows XP Service Pack 2 Firewall Settings

Complete the following steps to configure the Windows Firewall to allow the `REngine.exe` application to communicate with the TestStand client.

1. Log in as a user with administrator privileges.
2. Navigate to **Windows Firewall** on the Windows Control to launch the Windows Firewall dialog box.
3. Click **Off** on the General tab of the Windows Firewall dialog box to disable the firewall, or complete the following steps to add exceptions for the `REngine.exe` application with the firewall enabled.
 - a. Click the **Exceptions** tab.
 - b. Click the **Add Program** button to launch the Add a Program dialog box. Click **Browse** and select `<TestStand>\bin\REngine.exe`. Click **OK** to close the Add a Program dialog box.
 - c. Click the **Add Port** button to launch the Add a Port dialog box. In the **Name** control, type `DCOM`. In the **Port Number** control, type `135`. Select the **TCP** radio button and click **OK** to close the Add a Port dialog box.
4. Click **OK** to close the Windows Firewall dialog box.

Windows Vista Firewall Settings

Complete the following steps to configure the Windows Firewall to allow the `REngine.exe` application to communicate with the TestStand client.

1. Log in as a user with administrator privileges.
2. Navigate to **Windows Firewall** on the Windows Control Panel and select the **Change settings** option in the Windows Firewall window to launch the Windows Firewall Settings dialog box.

3. Complete the following steps to add exceptions for the `REngine.exe` application with the firewall enabled.
 - a. Click the **Exceptions** tab.
 - b. Click the **Add program** button to launch the Add a Program dialog box. Click **Browse** and select `<TestStand>\bin\REngine.exe`. Click **OK** to close the Add a Program dialog box.
 - c. Click the **Add port** button to launch the Add a Port dialog box. In the **Name** control, type `DCOM`. In the **Port Number** control, type `135`. Select the **TCP** radio button and click **OK** to close the Add a Port dialog box.
4. Click **OK** to close the Windows Firewall Settings dialog box.

Windows 2000 Service Pack 4

Complete the following steps to configure the security permissions for the server on a remote computer.

1. Log in as a user with administrator privileges.
2. Run `dcomcnfg` from the command line to launch the Distributed COM Configuration Properties dialog box.
3. Click the **Default Security** tab and specify the default security.
4. On the **Default Properties** tab, enable the **Enable Distributed COM on this computer** option.



Note You must restart the computer for changes to the value of the **Enable Distributed COM on this computer** option to take effect.

5. On the **Applications** tab, select **NI TestStand Remote Engine** and click the **Properties** button to launch the NI TestStand Remote Engine Properties dialog box. Click the **Security** tab to configure the permissions for a specific server and to give individual users access to the server.



Note You can grant access permission to the remote computer to everyone but grant launch permissions only to appropriate users because launch permissions allow access to the TestStand server on the remote computer.

6. On the **Identity** tab of the NI TestStand Remote Engine Properties dialog box, select the **Interactive User** option.
7. Click **OK** to close the dialog box.

Database Logging and Report Generation

TestStand can log results of a sequence execution to a database and generate reports in multiple formats. You must have a working knowledge of database concepts, SQL, and database management system (DBMS) client software to understand the TestStand database concepts in this chapter.

Database Concepts

Review the following key concepts for using databases with TestStand and the following key Windows features TestStand uses to communicate with a DBMS.

Databases and Tables

A database is an organized collection of data, where you can store and retrieve information. Most modern DBMSs, also known as database servers, store data in tables.

Tables contain records, also known as rows. Each row contains fields, also known as columns. Every table in a database must have a unique name, and every column within a table must have a unique name. Each column in a table has a data type, which varies depending on the DBMS. For example, Table 6-1 contains columns for the UUT number, a step name, a step result, and a measurement.

Table 6-1. Example Database Table

UUT_NUM	STEP_NAME	RESULT	MEAS
20860B456	TEST1	PASS	0.5
20860B456	TEST2	PASS	(NULL)
20860B123	TEST1	FAIL	0.1

Table 6-1. Example Database Table (Continued)

UUT_NUM	STEP_NAME	RESULT	MEAS
20860B789	TEST1	PASS	0.3
20860B789	TEST2	PASS	(NULL)

A row can contain an empty column value, which means the specific cell contains a NULL value, also referred to as an SQL NULL value.

The order of the data in the table is not important. Ordering, grouping, and other manipulations of the data occur when you retrieve the data from the table. Use an SQL SELECT command, or query, to retrieve records from a database. The query defines the content and order of the data you want to retrieve. The result of a query is called a record set or SQL Statement data. You can retrieve certain columns and rows from one table, or you can retrieve data from multiple tables. You can refer to each column you retrieve by the name of the column or by a one-based number that refers to the order of the column in the query.

Database Sessions

Database operations occur within a database session. A simple session uses the following order of operations:

1. Connect to the database.
2. Open database tables.
3. Retrieve data from and store data in the open database tables.
4. Close the database tables.
5. Disconnect from the database.

Microsoft ADO, OLE DB, and ODBC Database Technologies

TestStand uses Microsoft ActiveX Data Objects (ADO) as its database client technology. ADO, which is built on top of the Object Linking and Embedding Database (OLE DB), is one of several database interface technologies integrated into Windows operating systems.

Applications that use ADO, such as TestStand, use the OLE DB interfaces indirectly. The OLE DB layer interfaces to databases directly through a specific OLE DB provider for the DBMS or through a generic Open Database Connectivity (ODBC) provider, which interfaces to a specific ODBC driver for the DBMS. Figure 6-1 shows the high-level relationships between TestStand and components of Windows database technologies.

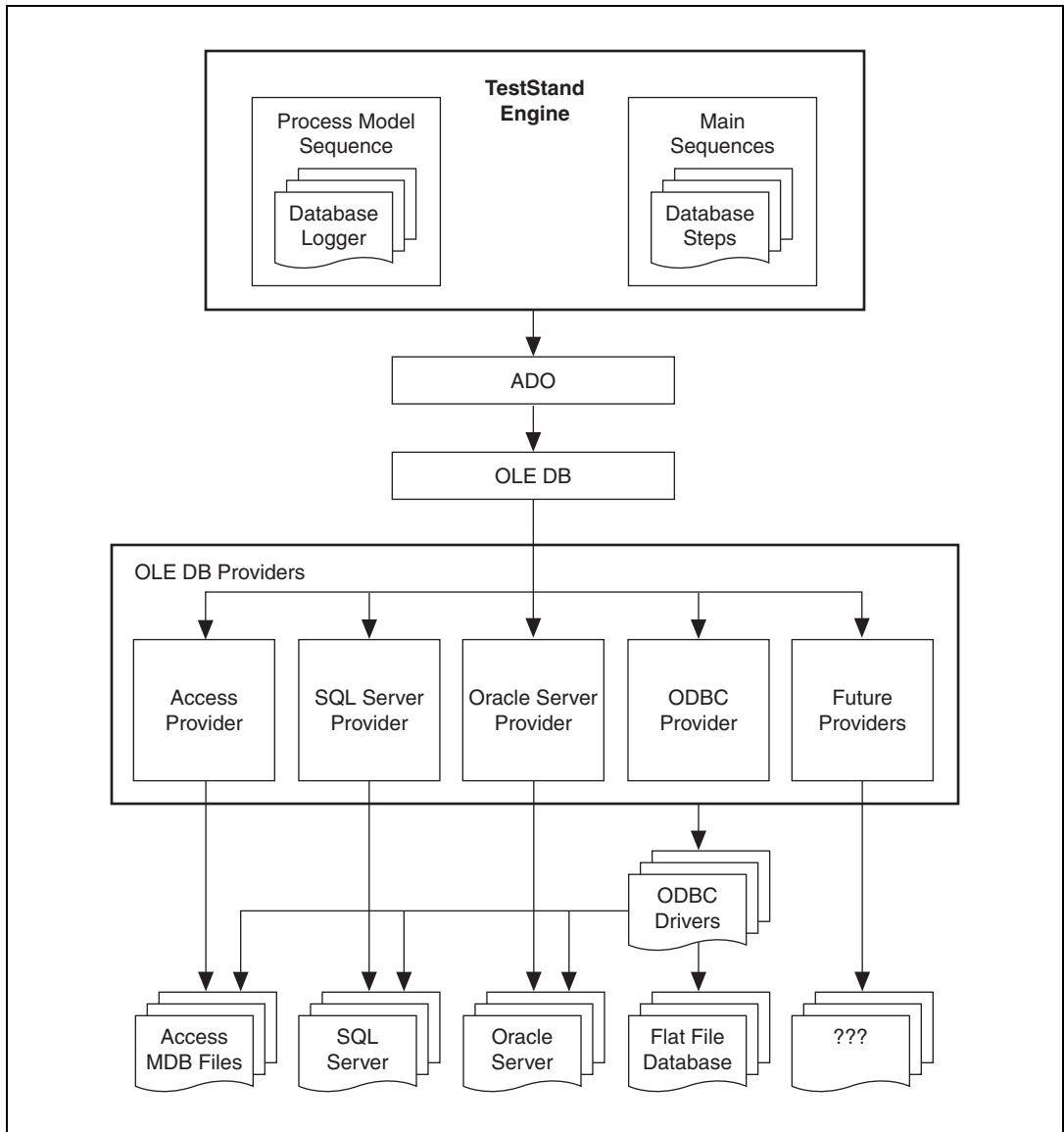


Figure 6-1. Microsoft Windows Database Technologies

Refer to www.microsoft.com for more information about database technologies for Windows operating systems.

Data Links

Before you can access data from a database within TestStand, you must use a data link to specify the server on which the data resides, the database or file that contains the data, the user ID, and the permissions to request when connecting to the data source.

For example, to connect to a Microsoft SQL Server database, specify the OLE DB provider for an SQL Server, a server name, a database name, a user ID, and a password. To connect to a Microsoft Access database, specify Microsoft Jet or specify the OLE DB provider for ODBC and an ODBC data source name. The ODBC data source name specifies which ODBC driver to use, the database file (.mdb), and an optional user ID and password. Use the ODBC Administrator on the Windows Control Panel to define ODBC data source names. Refer to the [Using the ODBC Administrator](#) section of this chapter for more information about the ODBC Administrator.

A connection string is a string version of the connection information in the data link required to open a session to a database. Use the Data Link Properties dialog box to build a connection string. The Data Link Properties dialog box and the information contained in the connection string vary according to the OLE DB provider. For example, a connection string for an SQL Server database might contain the following information:

```
Provider=SQLOLEDB.1;Integrated Security=SSPI;Persist
Security Info=True;User ID=guest;Initial
Catalog=pubs;Data Source=SERVERCOMPUTER
```

Complete the following steps to store the contents of a connection string in a Microsoft Data Link file (.udl).

1. Create a Data Link file by right-clicking in Windows Explorer and selecting **New»Text Document**.
2. Change the file extension to .udl.
3. Right-click the new file and select **Open** to launch the Data Link Properties dialog box.
4. Click the **Connection** tab and enable the **Use connection string** option.
5. Click the **Build** button to launch the Select Data Source dialog box, in which you can build a connection string. When you finish, click **OK** to close the Select Data Source dialog box.

6. Use the other options on the Provider, Connection, Advanced, and All tabs to provide additional configuration information for the .udl file.
7. Click **OK** to close the Data Link Properties dialog box. The .udl file automatically saves when you exit the dialog box.

Refer to the [Using Data Links](#) section of this chapter for more information about specifying data links. Refer to the *NI TestStand Help* for more information about the Data Link Properties dialog box.

Database Logging Implementation

The database logging capability is not native to the TestStand Engine or the TestStand Sequence Editor. The default process model contains customizable sequences that implement the database logging features. You can also customize or replace any portion of the database logging sequences. Refer to Appendix A, [Process Model Architecture](#), for more information about customizing the default process model.

The default process model, which calls the Main sequence in the client sequence file to test a UUT, relies on the automatic result collection capabilities of the TestStand Engine to accumulate the raw data to log to a database for each UUT. The engine automatically compiles the result of each step into a result list for an entire sequence, which contains the result of each step and the result list of each subsequence call it makes. Refer to the [Result Collection](#) section of Chapter 3, [Executions](#), for more information about automatic result collection.

The Test UUTs and Single Pass Execution entry points in the TestStand process models log the raw results to a database. By default, the Test UUTs entry point logs results after each pass through the UUT loop.

Select **Configure»Database Options** to launch the Database Options dialog box, in which you can specify the following options:

- The data link connection string TestStand uses to log results.
- The database schema TestStand uses. A schema contains the SQL statements, table definitions, and TestStand expressions that instruct TestStand how to log results to a database. TestStand includes a set of predefined schemas, which contains at least one schema for each supported DBMS. You can also create new schemas that log results to tables you define.

- Filtering options to limit the amount of data TestStand logs.
- If the process models log data after executing each step or after passing through each UUT loop.

Refer to the *NI TestStand Help* for more information about the Database Options dialog box.

Using Database Logging

Complete the following steps before you use the default process model to log results to a database.

1. Decide which DBMS you want to use. By default, TestStand supports SQL Server, Oracle, Access, Sybase, and MySQL. Refer to the [Adding Support for Other Database Management Systems](#) section of this chapter if you want to use another DBMS.
2. Make sure you installed the appropriate client DBMS software required to communicate with the DBMS.

You must decide to use an ODBC driver or a specific OLE DB provider for the DBMS. Use the OLE DB providers for SQL Server and Access. Most Oracle ODBC drivers and OLE DB providers require that you install Oracle Client.

Refer to the *Recommended Database Client Software* section of the *NI TestStand Release Notes* for more information about suggested providers, versions of ODBC drivers, client DBMS software, and any known issues.

3. Create the default database tables in the DBMS.

The <TestStand>\Components\Models\TestStandModels\Database directory contains SQL script files to create and delete the default database tables the default TestStand schemas require. For example, the Access Create Generic Recordset Result Tables.sql file contains SQL commands to create the default tables for Access. The Access Drop Result Tables.sql file contains SQL commands to delete the default tables.

TestStand includes an example Access database, TestStand Results.mdb, in the <TestStand>\Components\Models\TestStandModels\Database directory.

Refer to the [Database Viewer—Creating Result Tables](#) section of this chapter for more information about creating the default database tables using an SQL script file. Refer to the [Default TestStand Table Schema](#)

section of this chapter for more information about the default table schema the process model uses.

4. Use the Database Options dialog box to enable database logging and to define a data link and schema for the default process model to use.

Refer to the *NI TestStand Help* for more information about the Database Options dialog box. Refer to the [Using Data Links](#) section of this chapter for more information about defining data links.

Logging Property in the Sequence Context

When TestStand starts logging data to a database, it creates a temporary Logging property in the sequence context to evaluate expressions. The Logging property contains subproperties that provide information about database settings, process model data structures, and the results TestStand processes. As the Logging property processes the result list, TestStand updates the subproperties of the Logging property to refer to the UUT result, step result, and the step result subproperty TestStand is processing. You can reference the Logging subproperties in the precondition and value expressions you specify for schema statements and columns.

The Logging property contains the following subproperties:

- **UUTResult**—Contains the UUT result TestStand is processing. If TestStand is processing a step or a subproperty, this property holds the UUT result that contains the step result or subproperty.
- **StepResult**—Contains the step result TestStand is processing. If TestStand is processing a subproperty, this property holds the step result that contains the subproperty. If TestStand is processing a UUT result, this property contains the result of the sequence call in the process model that calls the Main sequence in the client file.
- **PropertyResult**—Contains the subproperty of the step result TestStand is processing. If TestStand is not processing a subproperty, this property does not exist.
- **PropertyResultDetails**—Contains information about the subproperty of the step result TestStand is processing. If TestStand is not processing a subproperty, this container does not exist.
- **ExecutionOrder**—Contains a numeric value TestStand increments after it processes each step result.
- **StartDate**—Specifies the date on which the UUT test began. This property is an instance of the DateDetails custom data type.

- **StartTime**—Specifies the time at which the UUT test began. This property is an instance of the TimeDetails custom data type.
- **UUT**—Specifies the serial number, test socket index, and other information about the UUT. This property is an instance of the UUT custom data type.
- **DatabaseOptions**—Contains the process model database settings you configure in the Database Options dialog box. This property is an instance of the DatabaseOptions custom data type.
- **StationInfo**—Specifies the station ID and the user name. This property is an instance of the StationInfo custom data type.

The TestStand process model files define the structure of the DateDetails, TimeDetails, UUT, DatabaseOptions, and StationInfo custom data types.

TestStand Database Result Tables

You can use the default table schemas, modify the existing schemas, or create new schemas.

Default TestStand Table Schema

The default TestStand database schema requires the following database tables:

- UUT_RESULT
- STEP_RESULT
- STEP_SEQCALL
- PROP_RESULT
- PROP_BINARY
- PROP_ANALOGWAVEFORM
- PROP_DIGITALWAVEFORM
- PROP_NUMERICLIMIT

The UUT_RESULT table contains information about each UUT TestStand tests. The STEP_RESULT table contains information about each step TestStand executes while testing each UUT. The STEP_SEQCALL table contains the sequence a Sequence Call step calls. The PROP_RESULT table contains information about the properties in a step result. The other table names with the PROP prefix contain information about specific property data types.

Each table contains a primary key column ID and might contain foreign key column IDs. The column data types are Number, String, or GUID depending on the schema. The column data types must match the primary key the data types reference.

Refer to the *NI TestStand Help* for more information about each TestStand database table.

Creating Default Result Tables with the Database Viewer

Use the TestStand Database Viewer application, which is located at `<TestStand>\Components\Tools\DatabaseView\DatabaseView.exe`, to create the default result tables the schema requires. You can also use the Database Viewer application to view data in a database, edit table information, and execute SQL commands.



Note To use the Database Viewer application, you must have previously set up the DBMS server and any required DBMS client software.

Refer to the *NI TestStand Help* for more information about the Database Viewer application. Refer to the [Database Viewer—Creating Result Tables](#) section of this chapter for more information about creating the default database tables using an SQL script file. Refer to the *NI TestStand Help* and to the [Using Data Links](#) section of this chapter for more information about configuring a computer to access the DBMS.

Adding Support for Other Database Management Systems

You can add support for DBMSs other than SQL Server, Oracle, Access, Sybase, and MySQL by adding a new schema in the Database Options dialog box or by using SQL scripts.

Use the Duplicate button on the Schemas tab of the Database Options dialog box to copy an existing schema and then customize its statement, column, and parameter settings to work with the new DBMS. The TestStand schemas for each DBMS conform to the default database tables.

Alternatively, you can create result tables for the default table schema for a similar DBMS by using the SQL script files located in the `<TestStand>\Components\Models\TestStandModels\Database` directory and modifying the schema for the new DBMS.

You can also complete the following steps to create new script files for a DBMS.

1. Create new script files in the <TestStand Public>\Components\Models\TestStandModels\Database directory. National Instruments recommends including the name of the DBMS in the filename.
2. In the new script files, enter the SQL commands for creating and deleting DBMS tables. Refer to the SQL database script files TestStand provides for guidelines. For example, the SQL database syntax file for Oracle result tables might contain the following commands for creating a UUT_Result table:

```
CREATE TABLE UUT_RESULT
(
    ID                                NUMBER PRIMARY KEY,
    UUT_SERIAL_NUMBER                CHAR (255) ,
    USER_LOGIN_NAME                  CHAR (255) ,
    START_DATE_TIME                  DATE ,
    EXECUTION_TIME                   NUMBER ,
    UUT_STATUS                       CHAR (255) ,
    UUT_ERROR_CODE                   NUMBER ,
    UUT_ERROR_MESSAGE                CHAR (255)
)
/
CREATE SEQUENCE SEQ_UUT_RESULT START WITH 1
/
CREATE FUNCTION UUT_RESULT_NEXT_ID RETURN NUMBER IS
    X NUMBER;
BEGIN
    SELECT SEQ_UUT_RESULT.NextVal INTO X FROM DUAL;
    RETURN X;
END;
/
```



Note Notice that the script uses three separate commands, each separated by the " / " character, to create the UUT_RESULT table in Oracle.

Use a similar syntax for deleting tables. For example, the SQL script file for Oracle might contain the following commands for deleting a UUT_RESULT table:

```
DROP TABLE UUT_RESULT
/
DROP SEQUENCE SEQ_UUT_RESULT
/
DROP FUNCTION UUT_RESULT_NEXT_ID
/
```

On-the-Fly Database Logging

When you enable the Use On-The-Fly Logging option in the Database Options dialog box, the process models progressively log result data concurrently with the execution instead of waiting until the execution or UUT test completes. Database logging uses the ProcessModelPostResultListEntry and SequenceFilePostResultListEntry callbacks to process the step results. The final data TestStand logs is almost identical to the data the process model generates at the end of execution.

When you use this option, you can use the Database Viewer application to view the data in the database tables while the sequence executes. Use the **Discard Results or Disable Results When Not Required by Model** option in the Model Options dialog box to conserve memory by discarding step results after TestStand logs each result.

If you use on-the-fly database logging with a schema that uses a stored procedure or command statements that do not use the INSERT command, you cannot define constraints for foreign keys in step result statements that reference primary keys in UUT results. Defining constraints for these types of foreign keys generates an error because the on-the-fly database logger cannot execute the statement to create the record that contains the primary key before executing the statement to create the record that contains the foreign key.

Using Data Links

You must define a data link when you specify the database where TestStand logs results or when you use the Database step types. Use the Data Link Properties dialog box to create or edit a data link connection string and to specify initialization properties for an OLE DB provider.

Refer to the *NI TestStand Help* for more information about the Data Link Properties dialog box.

Using the ODBC Administrator

To access databases through the ODBC standard, you must have an ODBC driver for each database system you use. Each ODBC driver must register itself with the operating system when you install it. You must also define and name data sources in the ODBC Administrator on the Windows Control Panel, which typically requires information such as a server, database, and additional database-specific options. You can define one or more data sources for each ODBC driver. Navigate to **Administrative Tools** on the Windows Control Panel and select **Data Sources (ODBC)** to launch the ODBC Administrator.

(Windows Vista) If you use the ODBC Administrator on Windows Vista and you receive any warnings, follow the prompts. You might need administrator access to create a new Database Source Name (DSN).



Note Because the database features of TestStand comply with the ODBC standard, you can use any ODBC-compliant database drivers. TestStand does not install any ODBC database drivers. DBMS vendors and third-party developers offer their own drivers. Refer to the vendor documentation for information about registering the specific database drivers with the ODBC Administrator.

Refer to the *NI TestStand Help* for more information about the ODBC Data Source Administrator dialog box.

Example Data Link and Result Table Setup for Microsoft Access

Use the following sections as an example of how to link a TestStand data link to an Access database file (.mdb) using the Jet OLE DB provider to log results using the default process model.

Database Options—Specifying a Data Link and Schema

Complete the following steps to configure the database logging options.

1. Launch the sequence editor and log in as Administrator.
2. Select **Configure»Database Options** to launch the Database Options dialog box. The Logging Options tab is active.
3. Enable database logging by removing the checkmark in the **Disable Database Logging** option.
4. Click the **Data Link** tab of the Database Options dialog box and select **Access** from the Database Management System ring control.
5. Click the **Build** button to launch the Data Link Properties dialog box.

6. Select **Microsoft Jet 4.0 OLE DB Provider** on the **Provider** tab of the Data Link Properties dialog box and click **Next**.
7. On the **Connection** tab, click **Browse** to launch the Select Access Database dialog box.
8. Using the Select Access Database dialog box, locate an Access database file (.mdb) and click **Open** to select the file.
9. In the Data Link Properties dialog box, click the **Test Connection** button to verify that you properly entered the required information.
10. Click **OK** to close the Data Link Properties dialog box.

Notice that the Connection String Expression in the Database Options dialog box now contains a literal string expression version of the data link connection string.

Database Viewer—Creating Result Tables

Complete the following steps to create the default result tables in a database.

1. If you are continuing from the steps in the previous section, skip to step 2. Otherwise, complete the following steps.
 - a. Launch the sequence editor and log in as Administrator.
 - b. Select **Configure»Database Options** to launch the Database Options dialog box. The Logging Options tab is active.
 - c. Enable database logging by removing the checkmark in the **Disable Database Logging** option.
 - d. Click the **Data Link** tab of the Database Options dialog box.
2. Click the **View Data** button to launch the Database Viewer application and open the data link.



Note The Connection String Expression must contain a valid expression to launch the Database Viewer application.

3. In the Database Viewer application, select **File»New Execute SQL Window** to open an Execute SQL window.
4. Click the **Load SQL Commands From File** button, select `<TestStand>\Components\Models\TestStandModels\Database\Access Create Generic Recordset Result Tables.sql`, and click **Open**.

Notice that the SQL Commands control now contains a set of SQL commands for creating the default result tables.

5. Click the **Execute SQL Commands** button to create the default result tables. Review the results of the SQL commands in the SQL History control to ensure that you created the tables successfully.
6. Click the Data Link window and select **Window»Refresh** to view the tables.

After you have completed these steps, any execution you launch with the Test UUTs or Single Pass entry point automatically logs its results to the database.

Test Report Implementation

Most of the test report capabilities are not native to the TestStand Engine or the TestStand Sequence Editor. The default process model contains customizable sequences that implement the test report features. You can also customize or replace any portion of the test reporting sequences. Refer to Appendix A, *Process Model Architecture*, for more information about customizing the default process model.

The default process model, which calls the Main sequence in the client sequence file to test a UUT, relies on the automatic result collection capabilities of the TestStand Engine to accumulate the raw data for each UUT test report. The engine automatically compiles the result of each step into a result list for an entire sequence, which contains the result of each step and the result list of each subsequence call it makes. Refer to the *Result Collection* section of Chapter 3, *Executions*, for information about automatic result collection.

You can also use the Report Options dialog box to customize the content of test reports. Refer to the *NI TestStand Help* for more information about the Report Options dialog box.

Using Test Reports

The Test UUTs and Single Pass entry points in the TestStand process models generate UUT test reports. The Test UUTs entry point generates a test report and writes it to disk after each pass through the UUT loop. Select **Configure»Report Options** to launch the Report Options dialog box, in which you can set options that determine the contents and format of the test report and the names and locations of test report files.

In the TestStand Sequence Editor, the Report tab on the Execution window displays the report for the current execution. Usually, the Report pane is empty until execution completes. The default process model generates reports in XML, HTML, or ASCII-text formats. You can also use an external application to view reports in these or other formats. Select **Configure»External Viewers** to specify the external application TestStand launches to display a particular report format. In the Execution window, click the **Viewer** button on the Report pane to view the report in the external viewer you specified.

Refer to the *NI TestStand Help* for more information about the Report pane, the Report Options dialog box, and the Configure External Viewers dialog box.

Failure Chain in Reports

For UUTs that fail, XML, HTML, and ASCII-text reports include a failure chain section in the report header. The first item in the failure chain table shows the step failure that caused the UUT to fail. The remaining items show the Sequence Call steps through which the execution reached the failing step. In XML and HTML reports, each step name in the failure chain links to the section of the report that displays the result for the step.

Batch Reports

When you use the Batch process model, the model generates a Batch report in addition to a report for each UUT. The batch report summarizes the results for all the UUTs in the batch. XML and HTML reports link to each UUT report.

Property Flags that Affect Reports

Set the PropFlags_IncludeInReport, PropFlags_IsLimit, and PropFlags_IsMeasurementValue flags to identify the result properties to automatically display in the report.

The IncludeInReport flag specifies to include a property in the report. For properties that hold limit values or output values, use the IsLimit and IsMeasurementValue flags to selectively exclude limits or output values according to the options you select in the Report Options dialog box. If you set a reporting flag for an array or container property, TestStand sets the flag for all array elements or subproperties within the container property.

On-the-Fly Report Generation

When you enable the On-The-Fly Reporting option on the Contents tab of the Report Options dialog box, the process models progressively generate the test report concurrently with the execution instead of waiting until the execution or UUT test completes. The final test report TestStand generates is identical to the test report the process model generates at the end of execution. You can use on-the-fly reporting only for HTML and ASCII reports.

When you use on-the-fly reporting, you can select the Report tab in the Execution window to view the test report during the execution. If the Report tab is the active view of the Execution window while a sequence executes, the test report periodically updates as TestStand processes step results.

In addition to generating the test report concurrently with execution, on-the-fly reporting periodically persists the current test report to a temporary file based on the persistence interval the process model sequences specify. TestStand deletes the temporary file and saves the final test report to a file at the end of a UUT loop execution. Refer to Appendix A, *Process Model Architecture*, for more information about process model sequences.

If you enable the Conserve Memory and Only Display Latest Results report option, on-the-fly reporting periodically purges internal data structures. As a result, the test report TestStand displays in the Report pane of the Execution window shows only the results for the steps on-the-fly reporting has not yet purged. The persisted temporary and final test report files contain all the step results. For these files, the step results for Sequence Call and Loop Result steps appear after the corresponding Sequence Call and Loop Index step results, respectively.

Use the **Discard Results or Disable Results When Not Required By Model** option in the Model Options dialog box to conserve memory by discarding step results after TestStand reports each result.

XML Report Schema

XML Test Reports conform to the XML W3C Schema. The XML Schema Definition (XSD) file is located at <TestStand>\Components\Models\TestStandModels\Report.xsd.

User Management

Use the TestStand User Manager to maintain the list of users, user names, user passwords, user privileges, groups, group privileges, and members of groups. TestStand can limit the functionality of the TestStand Sequence Editor and User Interfaces depending on the privilege settings you define in the user manager for the current user and the groups to which the user belongs.

By default, the sequence editor and the user interfaces launch the Login dialog box when you run TestStand.

Use the User Manager tab of the Station Options dialog box to specify if TestStand enforces user privileges and to specify the location of the user manager configuration file.



Note The TestStand User Manager helps you implement policies and procedures that concern the use of test stations. The user manager is not a security system, and it does not inhibit or control the operating system or third-party applications. Use the system-level security features your operating system provides to secure test station computers against unauthorized use.

Refer to the *NI TestStand Help* for more information about the User Manager tab of the Station Options dialog box, the User Manager window, adding groups and users, and setting privileges.

Privileges

The TestStand User Manager stores user and group privileges as Boolean properties and organizes the privileges in the following categories:

- **Operate**—Contains privileges for executing sequences and for terminating and aborting executions.
- **Debug**—Contains privileges for controlling execution flow, for executing manual and interactive executions, and for editing station globals and run-time variables.
- **Develop**—Contains privileges for editing and saving sequence files, for editing workspace files, and for using source code control.

- **Configure**—Contains privileges for configuring station options, user management, adapters, application settings, reports, database logging options, model options, and for editing process model files.
- **Custom**—Contains custom privileges you define. Customize the `NI_UserCustomPrivileges` data type to add new privileges.

You can grant all privileges in a specific category for each user or group in the user manager, and you can grant specific privileges for each user or group. In addition, when you add a user as a member of a group, TestStand grants the user all the privileges of the group. TestStand grants a privilege to a user or group if the property value for the privilege is `True` or if the value of the `GrantAll` property in any enclosing parent privilege category is `True`. For example, a user has the privilege to terminate an execution if one of the following properties is `True`:

- `<User>.Privileges.Operate.Terminate`
- `<User>.Privileges.Operate.GrantAll`
- `<User>.Privileges.GrantAll`
- `<Group>.Privileges.Operate.Terminate`
- `<Group>.Privileges.Operate.GrantAll`
- `<Group>.Privileges.GrantAll`

TestStand also grants all privileges to a user when you disable privilege checking on the User Manager tab of the Station Options dialog box.

Accessing Privilege Settings for the Current User

Call the `CurrentUserHasPrivilege` expression function to verify in an expression that the current user has a specific privilege.

Use the `Engine.CurrentUserHasPrivilege` method in the TestStand API to verify the privilege in a code module. The `Engine.CurrentUserHasPrivilege` method behaves identically to the `CurrentUserHasPrivilege` expression function.

When you call the `CurrentUserHasPrivilege` method or expression function, you must specify the property name of the privilege as a string argument. You can pass any subset of the property name tree structure to `CurrentUserHasPrivilege`. For example, you can call

CurrentUserHasPrivilege with the following expressions to determine if the current user has the privilege to terminate an execution:

- `CurrentUserHasPrivilege("Terminate")`
- `CurrentUserHasPrivilege("Operate.Terminate")`

You can pass "*" as the string argument to CurrentUserHasPrivilege to determine if a user is currently logged in. Refer to the [Expressions](#) section of Chapter 1, *NI TestStand Architecture*, for more information about using expressions. Refer to the *NI TestStand Help* for more information about the `Engine.CurrentUserHasPrivilege` method and the `CurrentUserHasPrivilege` expression function.

Accessing Privilege Settings for Any User

The TestStand API includes methods to access the privileges of any user or group. Use the `Engine.GetUser` and `Engine.GetUserGroup` methods to return a `User` object, then call the `User.HasPrivilege` method, which returns `True` if the user or any group to which the user belongs has the privilege you specify by name. When you call the `User.HasPrivilege` method on a `User` object the `Engine.GetUser` method returns, the `User.HasPrivilege` method behaves identically to the `CurrentUserHasPrivilege` method or expression function. Refer to the *NI TestStand Help* for more information about the `User` object and its methods.

Defining Custom Privileges

By default all users and groups include an empty `Privileges.Custom` category. Add Boolean properties to the `NI_UserCustomPrivileges` standard data type in the User Manager file in the Types window to define new privileges in the category. When you add new properties to the data type, increment the version of the type to remove the modified flag and to ensure that TestStand uses the modified type instead of the default type TestStand installs.

The sequence editor and user interfaces that use the TestStand User Interface (UI) Controls do not recognize custom privileges you define. You must add code to user interfaces to handle the custom privileges you create. For example, you can add a calibration operation to the user interface where you want to define a custom privilege so only specific users can perform the operation.

Customizing and Configuring TestStand

You can customize TestStand User Interfaces, process models, callbacks, data types, step types, the Tools menu, and the directory structure. You can configure options for the sequence editor, user interfaces, and test station.

Other chapters in this manual describe user interfaces, process models, callbacks, data types, and step types in greater detail. Table 8-1 includes a brief description of how you can modify these components and where you can find more information about the component.

Table 8-1. TestStand Customizable Components

Component	How to Customize	More Information
User Interfaces	TestStand includes full source code in several different programming languages so you can modify the user interfaces to meet specific needs.	Refer to Chapter 9, <i>Creating Custom User Interfaces</i> .
Process Models	TestStand includes fully customizable Sequential, Parallel, and Batch process models to meet specific testing needs.	Refer to Chapter 10, <i>Customizing Process Models and Callbacks</i> .
Callbacks	You can modify callback sequences to customize the operation of test stations.	Refer to Chapter 10, <i>Customizing Process Models and Callbacks</i> .
Data Types	You can customize copies of the standard TestStand data types, and you can create and modify your own data types.	Refer to Chapter 11, <i>Type Concepts</i> , and Chapter 12, <i>Standard and Custom Data Types</i> .
Step Types	You can customize copies of the standard TestStand step types, and you can create and modify your own step types.	Refer to Chapter 11, <i>Type Concepts</i> , and Chapter 13, <i>Custom Step Types</i> .

Tools Menu

The Tools menu in the TestStand Sequence Editor and User Interfaces contains common tools for use with TestStand. You can modify the Tools menu to contain exactly the tools you need, and you can also add new items to the Tools menu. Refer to the *NI TestStand Help* for more information about the Tools menu and about using the Customize Tools Menu dialog box to add your own commands to the Tools menu.

TestStand Directory Structure

To comply with Windows Vista restrictions on writing to the Program Files directory and to improve usability for Windows XP users who do not have access to the Program Files directory, TestStand 4.1 installs some files in different locations from previous versions of TestStand. Refer to the *NI TestStand Release Notes* for more information about specific directories and files relocated in TestStand 4.1.

TestStand 4.1 installs files in the following directories:

- **<TestStand>**—Located by default at C:\Program Files\National Instruments\TestStand x.x on Windows 2000/XP and Windows Vista (32-bit) and at C:\Program Files (x86)\National Instruments\TestStand x.x on Windows Vista (64-bit).
- **<TestStand Public>**—Located by default 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.
- **<TestStand Application Data>**—Hidden by default and located at C:\Documents and Settings\All Users\Application Data\National Instruments\TestStand x.x on Windows 2000/XP and at C:\ProgramData\National Instruments\TestStand x.x on Windows Vista.

<TestStand> Directory

The <TestStand> directory is the location where you installed TestStand on the computer and contains the read-only TestStand program files. Table 8-2 shows the name and content of each subdirectory of the <TestStand> directory.

Table 8-2. <TestStand> Subdirectories

Directory Name	Contents
AdapterSupport	Read-only support files for the .NET and HTBasic Adapters.
API	Read-only TestStand ActiveX automation server libraries and utility libraries for several programming languages.
Bin	Read-only TestStand Sequence Editor executable, TestStand Engine DLLs, and support files.
CodeTemplates	Read-only source code templates for step types.
Components	Read-only components installed with TestStand, including callback files, converters, icons, language files, process model files, step types, source files, compatibility files, and utility files. Refer to the <i>Components Directory</i> section of this chapter for more information about the subdirectories in the Components directory.
Doc	Documentation files.
UserInterfaces	Read-only LabVIEW, LabWindows/CVI, Microsoft Visual Basic, C#, and C++ (MFC) user interfaces with source code.

Components Directory

TestStand installs the default sequences, executables, project files, and source files for TestStand components in the <TestStand>\Components directory. Table 8-3 lists each subdirectory in the <TestStand>\Components directory.

Table 8-3. TestStand Component Subdirectories

Directory Name	Contents
Callbacks	Contains the sequence files in which TestStand stores Station Engine callbacks and Front-End callbacks. Refer to Chapter 10, <i>Customizing Process Models and Callbacks</i> , for more information about customizing the Station Engine and Front-End callbacks.
Compatibility	Contains type palette files TestStand uses to save sequence files compatible with earlier versions of TestStand.
Icons	Contains icon files for module adapters and step types.

Table 8-3. TestStand Component Subdirectories (Continued)

Directory Name	Contents
Language	Contains string resource files in language-specific subdirectories. Refer to the Creating String Resource Files section of this chapter for more information about creating string resource files.
Models	Contains the default process model sequence files and supporting code modules. Refer to Chapter 10, Customizing Process Models and Callbacks , for more information about customizing process models.
Obsolete	Contains components TestStand no longer uses but installs to maintain backward compatibility.
RuntimeServers	Contains a LabVIEW 7.1.1 run-time application for executing LabVIEW 7.1.1-based code modules on a computer on which you have not installed the LabVIEW Development System. Refer to the Selecting a LabVIEW Server section of Chapter 5, Configuring the LabVIEW Adapter , of the Using LabVIEW with TestStand manual for more information about using LabVIEW executables built with an ActiveX server enabled.
StepTypes	Contains support files for step types. TestStand installs support files for the built-in step types in the <TestStand>\Components\StepTypes directory. Refer to Chapter 13, Custom Step Types , for more information about customizing step types.
Tools	Contains sequences and supporting files for the Tools menu commands. Refer to the Tools Menu section of this chapter for more information about customizing the Tools menu.
TypePalettes	Contains the default type palette files. Refer to Chapter 4, Built-In Step Types , for more information about the default step types. Refer to Chapter 11, Type Concepts , for more information about step types and data types.

The TestStand Engine searches for sequences and code modules using the TestStand search directory path. The default search precedence places the <TestStand Public>\Components directory before the <TestStand>\Components directory to ensure that TestStand loads the sequences and code modules you customize instead of loading the default TestStand versions of the files. Select **Configure»Search Directories** from the sequence editor menu bar to modify the precedence of the TestStand search directory paths. Refer to the [<TestStand Public> Directory](#) section of this chapter for more information about the

<TestStand Public> directory. Refer to the [Search Paths](#) section of Chapter 5, [Module Adapters](#), for more information about TestStand search directories.

When you deploy a run-time version of the TestStand Engine, you can bundle customized components in the <TestStand Public> directory with the TestStand run-time deployment. Refer to Chapter 14, [Deploying TestStand Systems](#), for more information about deploying the TestStand Engine with custom components.

<TestStand Public> Directory

The <TestStand Public> directory contains your modifications, customizations, and other files you can directly edit. Table 8-4 shows the name and content of each subdirectory of the <TestStand Public> directory.

Table 8-4. <TestStand Public> Subdirectories

Directory Name	Contents
AdapterSupport	Support files for the LabVIEW and LabWindows/CVI Adapters.
CodeTemplates	Empty by default. You can store modified source code templates for step types, or code templates you create.
Components	Placeholder subdirectories for modified TestStand components and components you develop, including callback files, language files, process model files, run-time servers, and type palette files. Refer to the Components Directory section of this chapter for more information about the subdirectories in the default Components directory TestStand installs. Refer to the RuntimeServers Directory section of this chapter for more information about run-time servers.
Examples	Example sequences and tests. Most example sequences that use LabVIEW VIs call subVIs in the <LabVIEW>\vi.lib directory, which you can access after you install LabVIEW.
Setup	Support files for the TestStand installer.

Table 8-4. <TestStand Public> Subdirectories (Continued)

Directory Name	Contents
Tutorial	Sequences and code modules you use in tutorials in this manual, the <i>Using TestStand</i> manual, the <i>Using LabVIEW with TestStand</i> manual, the <i>Using LabWindows/CVI with TestStand</i> manual, and the <i>NI TestStand Evaluation Guide</i> .
UserInterfaces	Copies of the LabVIEW, LabWindows/CVI, Microsoft Visual Basic, C#, and C++ (MFC) user interfaces with source code installed in the <TestStand>\UserInterfaces directory.

RuntimeServers Directory

The <TestStand Public>\Components\RuntimeServers directory contains a LabVIEW 7.1.1 run-time application for executing LabVIEW 7.1.1-based code modules on a computer on which you have not installed the LabVIEW Development System. Refer to the *Selecting a LabVIEW Server* section of Chapter 5, *Configuring the LabVIEW Adapter*, of the *Using LabVIEW with TestStand* manual for more information about using LabVIEW executables built with an ActiveX server enabled.

Copying Read-Only Files to Modify

By default, TestStand installs read-only source files in the <TestStand>\CodeTemplates, <TestStand>\Components, and <TestStand>\UserInterfaces directories. To modify the installed code templates or components or to create new code templates or components, copy the files from the <TestStand> directories to the <TestStand Public> directories and make changes to the copies of the files. To modify the installed user interfaces or to create new user interfaces, modify the files TestStand installs in the <TestStand Public>\UserInterfaces directory. When you copy installed files to modify, 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. The `<TestStand Public>` directory also acts as a temporary location for components you use to build a deployment.

<TestStand Application Data> Directory

TestStand 4.1 also installs the `<TestStand Application Data>` directory. This directory is hidden by default and contains configuration files users generally do not edit but a program can edit. The directory includes the `Cfg` subdirectory, which contains configuration files for TestStand Engine, TestStand Sequence Editor, and TestStand User Interface options.

Creating String Resource Files

TestStand uses the `Engine.GetResourceString` method to obtain the string messages to display in sequence editor and user interface windows and dialog boxes. The `Engine.GetResourceString` method uses a string category and a tag name as arguments and searches for the string resource in all string resource files in the following predefined order of directories:

1. `<TestStand Public>\Components\Language\<current language>`
2. `<TestStand Public>\Components\Language\English`
3. `<TestStand Public>\Components\Language`
4. `<TestStand>\Components\Language\<current language>`
5. `<TestStand>\Components\Language\English`
6. `<TestStand>\Components\Language`

Select **Configure»Station Options** to change the current language setting.

To customize a string resource file for a supported language or to create a resource file for a new language, copy an existing language file from the `<TestStand>\Components\Language\<language>` directory, place the file in the `<TestStand Public>\Components\Language\<language>` directory, and modify the file. To create a resource string file that applies to all languages, place the resource file in the base `<TestStand Public>\Components\Language` directory.



Note The TestStand Engine loads resource files when you start TestStand. If you make changes to the resource files, you must restart TestStand for the changes to take effect, or you must call the `Engine.ReloadStringResourceFiles` method.

String Resource File Format

String resource files must use the `.ini` file extension and use the following format:

```
[category1]
tag1 = "string value 1"
tag2 = "string value 2"

[category2]
tag1 = "string value 1"
tag2 = "string value 2"
```

When you create new entries in a string resource file or create a string resource file for custom components, use unique category names to avoid conflicts with the default names TestStand uses. For example, begin new category names with a unique ID, such as a company prefix.

You can create an unlimited number of categories and tag names. You can create strings of unlimited size, but you must break a string with more than 512 characters into multiple lines. Each line includes a `lineNNNN` tag suffix, where `NNNN` is the line number with zero padding, as shown in the following example:

```
[category1]
tag1 line0001 = "This is the first sentence of a long "
tag1 line0002 = "paragraph. This is the second sentence."
```

You can use the escape codes in Table 8-4 to insert unprintable characters.

Table 8-5. Resource String File Escape Codes

Escape Code	Description
<code>\n</code>	Embedded linefeed character.
<code>\r</code>	Carriage return character.
<code>\t</code>	Tab character.
<code>\xnn</code>	Hexadecimal value that represents the character. For example, <code>\x1B</code> represents the ASCII ESC character.
<code>\\</code>	Backslash character.
<code>\"</code>	Double quotation mark.

Configuring Sequence Editor and User Interface Startup Options

The sequence editor and all user interface applications support command-line options for opening and running sequences. You can append the startup options in Table 8-5 to the sequence editor and user interface command line. The “/” and “-” characters are valid command prefixes. Use spaces to separate command parameters. You must use quotation marks for arguments that contain spaces, such as "Test UUTs" and "C:\My Documents\MySeq.seq".

Table 8-6. Sequence Editor and User Interface Startup Options

Option	Purpose
<i>sequencefile</i> <i>{sequencefile2}...</i>	Instructs the application to automatically load the sequence files at startup, as shown in the following example: SeqEdit.exe "c:\My Seqs\seq1.seq" "c:\My Seqs\seq2.seq"
<i>/run sequence</i> <i>sequencefile</i>	Instructs the application to automatically load and run the sequence in the sequence file at startup, as shown in the following example: SeqEdit.exe /run MainSequence "c:\My Seqs\test.seq"
<i>/runEntryPoint</i> <i>entrypointname</i> <i>sequence file</i>	Instructs the application to automatically load and run the sequence file at startup using the Execution entry point you specify, as shown in the following example: SeqEdit.exe /runEntryPoint "Test UUTs" "c:\My Seqs\test.seq"
<i>/editor</i>	Instructs the application to open in Editor Mode if the application supports editing, as shown in the following example: testexec.exe /editor
<i>/operatorInterface</i>	Instructs the application to open in Operator Mode, as shown in the following example: testexec.exe /operatorInterface

Table 8-6. Sequence Editor and User Interface Startup Options (Continued)

Option	Purpose
/quit	<p>Instructs the application to exit after running the executions you specify, as shown in the following example:</p> <pre>SeqEdit.exe /run MainSequence "c:\My Seqs\test\seq" /quit</pre> <p>TestStand ignores the /quit option if the execution fails to launch.</p>
/useExisting	<p>Instructs the application to use the existing running instance of the application instead of opening a new instance, as shown in the following example:</p> <pre>SeqEdit.exe /useExisting</pre> <p>TestStand ignores the /useExisting option if you specify the /quit option.</p>
/setCurrentDir	<p>Instructs the application to set the current directory to the first directory in the File dialog box directory history list, as shown in the following example:</p> <pre>SeqEdit.exe /setCurrentDir</pre> <p>The current directory is the directory the File dialog box initially displays when you open or save a file. Use this option to set the directory the File dialog box displays to the directory the File dialog box displayed the last time you ran the application. TestStand sets the current directory after processing the other command-line options.</p>
/?	<p>Instructs the application to launch a help dialog box that contains a list of valid command-line arguments and then close immediately, as shown in the following example:</p> <pre>SeqEdit.exe /?</pre> <p>TestStand ignores all other options if you specify the "/" option.</p>

Refer to the `ApplicationMgr.ProcessUserCommandLineArguments` event documentation in the *NI TestStand Help* for more information about processing user-defined command-line arguments in a user interface. If an error exists in the command-line argument, the Application Manager control generates an `ApplicationMgr.ReportError` event.

Configure Menu

Use the Configure menu in the sequence editor and in the user interfaces to control the operation of the TestStand station. Refer to the *NI TestStand Help* for more information about the dialog boxes that each item in the Configure menu launches.

Creating Custom User Interfaces

You can create or customize a user interface application, including custom sequence editors and applications that can only run sequences.

Refer to the following documents before you create a custom user interface application:

- The *Example User Interfaces* section and the [Writing an Application with the TestStand UI Controls](#) section of this chapter.
- The following sections of the *NI TestStand Help*:
 - *TestStand ActiveX API Overview*
 - *Core UI Classes, Properties, Methods, and Events*
 - *Core API Classes, Properties, and Methods*
- The *NI TestStand User Interface Controls Reference Poster*.
- Chapter 6, *Creating Custom User Interfaces in LabVIEW*, of the *Using LabVIEW with TestStand* manual and Chapter 6, *Creating Custom User Interfaces in LabWindows/CVI*, of the *Using LabWindows/CVI with TestStand* manual.

If you use an environment other than LabVIEW or LabWindows/CVI, refer to one of these sources for general instructions for constructing a user interface application.

Example User Interfaces

The <TestStand>\UserInterfaces directory includes the executable, project, and source code files for each example user interface. The Full-Featured subdirectory contains user interfaces for loading, viewing, editing, saving, executing, and debugging sequence files. The Simple subdirectory contains similar but limited user interfaces with fewer commands and options but no menus. Also, the simple example user interfaces display the steps for executions you run but do not display steps for sequences you load. Both subdirectories contain source code for LabVIEW, LabWindows/CVI, Microsoft Visual Basic .NET, C#, and C++ (MFC).

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



Note National Instruments recommends that you track the changes you make to the user interface source code files so you can integrate the changes with any enhancements in future versions of the TestStand User Interfaces.

TestStand User Interface Controls

All user interface examples use the TestStand User Interface (UI) Controls, which are a set of ActiveX controls that implement the common functionality for applications to display, execute, edit, save, and debug test sequences. These ActiveX controls greatly reduce the amount of source code a user interface application requires and greatly reduce the need for an application to directly call the TestStand API.



Note National Instruments strongly recommends using the TestStand UI Controls to develop user interface applications.

You can create an application by directly calling the TestStand API on objects you create or obtain from the TestStand UI Controls properties, methods, or events. Consider the following guidelines when you call the TestStand API in a user interface that uses the TestStand UI Controls:

- You do not need to create the TestStand Engine. Use the `ApplicationMgr.GetEngine` method to obtain the `Engine` object.
- If you call the `Engine.NewExecution` method to create an execution, the TestStand UI Controls recognize the new execution.

- If you call the `Engine.GetSequenceFileEx` method to load a sequence file, the TestStand UI Controls do not display the file you load. You must call the `ApplicationMgr.OpenSequenceFile` method to open and display a file in the user interface.
- You can obtain sequence file and execution references from events or from the `SequenceFiles` and `Executions` collections.
- If you hold references to TestStand objects, release them in the handler for the `ApplicationMgr.QueryShutdown` event if the event handler does not cancel the shut down process.

Refer to the *Writing an Application with the TestStand Engine API* section of the *NI TestStand Help* for more information about writing an application by directly calling the TestStand Engine API.

Writing an Application with the TestStand UI Controls

TestStand provides manager controls and visible controls that work together to simplify programming a user interface.

Manager Controls

Application Manager, SequenceFileView Manager, and ExecutionView Manager controls call the TestStand API to perform tasks such as loading files, launching executions, and retrieving sequence and execution information. Manager controls also notify you when application events occur, such as when a user logs in, an execution reaches a breakpoint, or a user changes the file or sequence they are viewing. Manager controls are visible at design time and invisible at run time.

Connect the manager controls to visible TestStand UI Controls to display information or to allow users to select items to view.

Application Manager

The Application Manager control performs the following basic operations, which are necessary to use the TestStand Engine in an application:

- Processes command-line arguments.
- Maintains an application configuration file.
- Initializes and shuts down the TestStand Engine.
- Logs users in and out.
- Loads and unloads files.

- Launches executions.
- Tracks existing sequence files and executions.

An application must have a single Application Manager control that exists for the duration of the application.

SequenceFileView Manager

The SequenceFileView Manager control performs the following tasks to manage how other visible TestStand UI Controls view and interact with a selected sequence file:

- Designates a sequence file as the selected sequence file.
- Tracks which sequence, step group, and steps users select in the sequence file.
- Tracks which variables or properties users select in the sequence file.
- Displays aspects of the sequence file in the visible TestStand UI Controls to which the SequenceFileView Manager control connects.
- Enables visible TestStand UI Controls to which the SequenceFileView Manager control connects to change the selected file, sequence, step group, and steps.
- Provides editing and saving commands.
- Provides methods for executing the sequence file users select.

An application needs one SequenceFileView Manager control for each location, such as a window, form, or panel, in which you display a sequence file or let users select a sequence file.

ExecutionView Manager

The ExecutionView Manager control performs the following tasks to manage how other visible TestStand UI Controls view and interact with a selected TestStand execution:

- Designates an execution as the selected execution.
- Tracks which thread, stack frame, sequence, step group, and steps users select in the execution.
- Tracks which variables or properties users select in the execution.
- Displays aspects of the selected execution in the visible TestStand UI Controls to which the ExecutionView Manager control connects.

- Enables visible TestStand UI Controls to which the ExecutionView Manager control connects to change the selected thread, stack frame, sequence, step group, and steps.
- Sends events to notify the application of the progress and state of the execution.
- Provides debugging commands.
- Updates the ReportView control to show the current report for the execution.

An application needs one ExecutionView Manager control for each location, such as a window, form, or panel, in which you display an execution or let users select an execution.

Visible Controls

The TestStand UI Controls in Table 9-1 are visible at design time and run time and are similar to common Windows UI controls. Connect visible TestStand UI Controls to manager controls to display information or to allow users to select items to view.

Table 9-1. Visible TestStand UI Controls

Control Name	Description
Button	Connect a manager control to a Button control to specify that the button performs a common user interface command, such as “Open Sequence File.” The Button control uses a localized caption and automatically enables or disables according to the application state.
CheckBox	Connect a manager control to a CheckBox control so users can toggle the state of a common user interface command, such as “Break on Step Failure.”
ComboBox	Connect a manager control to a ComboBox control so users can view or select from a list of adapters, sequence files, sequences, step groups, executions, threads, or stack frames.

Table 9-1. Visible TestStand UI Controls (Continued)

Control Name	Description
ExpressionEdit	<p>Use an ExpressionEdit control so users can edit a TestStand expression with syntax coloring, popup help, and statement completion.</p> <p>Although you typically do not need to edit expressions in a user interface application, you can connect a manager control to a read-only ExpressionEdit control to display text information about the application state, such as the pathname of the selected sequence file or the name of the current user.</p> <p>You can also use ExpressionEdit controls in dialog boxes for step types and in tools in which you prompt users to enter a TestStand expression.</p>
InsertionPalette	Connect a SequenceFileView Manager control to an InsertionPalette control so users can insert steps and template items into a sequence file by dragging or double-clicking.
Label	Connect a manager control to a Label control to display text information about the application state in the label, such as the name of the current user or the status of the current UUT.
ListBar	Use a ListBar control to display multiple pages, where each page contains a list of items users can view or select. Connect a manager control to a ListBar page so users can view and select from a list of adapters, sequence files, sequences, step groups, executions, threads, or stack frames.
ListBox	Connect a manager control to a ListBox control so users can view or select from a list of adapters, sequence files, sequences, step groups, executions, threads, or stack frames.
ReportView	Connect an ExecutionView Manager control to a ReportView control to display the report for the selected execution.
SequenceView	Connect a SequenceFileView Manager control or an ExecutionView Manager control to a SequenceView control to display the steps of a sequence from a sequence file or execution. The SequenceView control displays the steps in a list with columns you specify when you configure the control.

Table 9-1. Visible TestStand UI Controls (Continued)

Control Name	Description
StatusBar	Connect a manager control to panes of a StatusBar control to display textual, image, or progress information about the application state. You can programmatically control individual StatusBar panes to display custom information.
VariablesView	Connect a SequenceFileView Manager control or an ExecutionView Manager control to a VariablesView control to display the sequence context for the sequence file or execution.

Connecting Manager Controls to Visible Controls

Connect a Manager control to a visible control to display sequences or reports, present a list of items to users, invoke an application command, or display information about the current state of the application. When you connect controls, you do not need to write the majority of the source code you usually write for the application to update the user interface and respond to user input.

You can make view connections, list connections, command connections, and information source connections, depending on the type of manager control and visible control you connect.

Refer to the *NI TestStand User Interface Controls Reference Poster* for an illustration of control connections in a sample user interface.

View Connections

You can connect manager controls to specific UI controls to display the steps in a sequence file or an execution, the report for an execution, the sequence context for a sequence file or execution, and the set of step types and templates users can insert into sequence files.

Connect a SequenceFileView Manager control or an ExecutionView Manager control to a SequenceView control to display the steps of a sequence from a sequence file or execution. You can also connect an ExecutionView Manager control to a ReportView control to display the report for the execution.

Connect a SequenceFileView Manager control or an ExecutionView Manager control to a VariablesView control to display the sequence context for the sequence file or execution.

Connect a `SequenceFileView Manager` control to an `InsertionPalette` control so users can insert steps and template items into a sequence file by dragging or double-clicking.

Call the following methods to connect to view controls:

- `SequenceFileViewMgr.ConnectSequenceView`
- `SequenceFileViewMgr.ConnectVariables`
- `SequenceFileViewMgr.ConnectInsertionPalette`
- `ExecutionViewMgr.ConnectExecutionView`
- `ExecutionViewMgr.ConnectReportView`
- `ExecutionViewMgr.ConnectVariables`

List Connections

You can connect a `ComboBox` control, a `ListBox` control, or a `ListBar` page to a list a manager control provides, as shown in Table 9-2.

Table 9-2. Available List Connections

List	Manager Control
Adapters	Application Manager
Sequence files	SequenceFileView Manager
Sequences	SequenceFileView Manager
Step groups	SequenceFileView Manager
Executions	ExecutionView Manager
Threads	ExecutionView Manager
Stack frames	ExecutionView Manager

A manager control designates one item in each list as the selected item. A visible control you connect to a list displays the list and indicates the selected item. The visible control also allows users to change the selection unless the application state or control configuration prohibits changing the selection. When users change the selection, other controls that display the list or the selected list item update to display the new selection. For example, you can connect a `SequenceFileView Manager` control to a `SequenceView` control and connect the sequence file list to a combo box. When users change the file selection in the combo box, the `SequenceView` control updates to show the steps in the newly selected sequence file.

Call the following methods to connect a list to a `ComboBox` control, a `ListBox` control, or a `ListBar` page:

- `ApplicationMgr.ConnectAdapterList`
- `SequenceFileViewMgr.ConnectSequenceFileList`
- `SequenceFileViewMgr.ConnectSequenceList`
- `SequenceFileViewMgr.ConnectStepGroupList`
- `ExecutionViewMgr.ConnectExecutionList`
- `ExecutionViewMgr.ConnectThreadList`
- `ExecutionViewMgr.ConnectCallStack`

Command Connections

TestStand applications typically use menus, buttons, or other controls to provide commands to users. The `OpenSequenceFile`, `ExecuteEntryPoint`, `RunSelectedSteps`, `Break`, `Resume`, `Terminate`, and `Exit` commands are common to most TestStand applications.

The `CommandKinds` enumeration in the TestStand UI Controls API defines a set of common commands you can add to an application. Refer to the *NI TestStand Help* for more information about this enumeration before you add commands to an application so you do not unnecessarily re-implement an existing command.

You can connect these commands to TestStand buttons or application menu items, which automatically execute the command. You do not need an event handler to implement the command.

The commands also determine the menu item or button text to display according to the current language and automatically dim or enable buttons or menu items according to the state of the application. Because the TestStand UI Controls API implements many common application commands, connecting commands to buttons and menu items significantly reduces the amount of source code an application requires.

Some commands apply to the selected item in the manager control to which you connect. For example, the `Break` command suspends the current execution an `ExecutionView Manager` control selects. Other commands, such as `Exit`, function the same regardless of the manager control you use to connect them.

Refer to the *NI TestStand Help* for more information about each `CommandKinds` enumeration constant and the manager controls to which the `CommandKinds` enumeration constant applies.

Call the following methods to connect a command to a `Button` or `CheckBox` control:

- `ApplicationMgr.ConnectCommand`
- `SequenceFileViewMgr.ConnectCommand`
- `ExecutionViewMgr.ConnectCommand`

Refer to the *Menus and Menu Items* section of this chapter for more information about connecting commands to menu items.

To invoke a command without connecting it to a control, call one of the following methods to obtain a `Command` object:

- `ApplicationMgr.GetCommand`
- `ApplicationMgr.NewCommands`
- `SequenceFileViewMgr.GetCommand`
- `ExecutionViewMgr.GetCommand`

After you obtain a `Command` object, call the `Command.Execute` method to invoke the command.

Information Source Connections

You can use manager controls to establish caption, image, and numeric value information source connections to `Label` controls, `ExpressionEdit` controls, and `StatusBar` panes to display information about the state of the application.

Caption Connections

Caption connections display text that describes the status of the application. For example, you can use the Application Manager control to connect a caption to a `Label` control so that the `Label` control displays the name of the currently logged-in user.

The `CaptionSources` enumeration defines the set of captions to which you can connect. Some captions apply to the selected item in the manager control with which you connect them. For example, the `UUTSerialNumber` caption displays the serial number of the current UUT for the execution an `ExecutionView Manager` control selects. Other captions, such as `UserName`, function the same regardless of which manager control you use to connect them.

Refer to the *NI TestStand Help* for more information about each `CaptionSources` enumeration constant and the manager controls with which the caption source functions.

Call the following methods to connect a caption to a `Label` control, an `ExpressionEdit` control, or a `StatusBar` pane:

- `ApplicationMgr.ConnectCaption`
- `SequenceFileViewMgr.ConnectCaption`
- `ExecutionViewMgr.ConnectCaption`

Call the following methods to obtain the text of a caption without connecting the caption to a control:

- `ApplicationMgr.GetCaptionText`
- `SequenceFileViewMgr.GetCaptionText`
- `ExecutionViewMgr.GetCaptionText`

Image Connections

Image connections display icons that illustrate the status of the application. For example, you can use the `ExecutionView Manager` control to connect an image to a `Button` control or a `StatusBar` pane so the button or pane displays an image that indicates the execution state of the selected execution.

The `ImageSources` enumeration defines the set of images to which you can connect. Some images apply to the selected item in the manager control with which you connect them. For example, the `CurrentStepGroup` enumeration constant displays an image for the currently selected step group when you connect it to a `SequenceFileView Manager` control and displays an image for the currently executing step group when you connect it to an `ExecutionView Manager` control.

Refer to the *NI TestStand Help* for more information about each `ImageSources` enumeration constant and the manager controls with which the image source functions.

Call the following methods to connect an image to a `Button` control or a `StatusBar` pane:

- `ApplicationMgr.ConnectImage`
- `SequenceFileViewMgr.ConnectImage`
- `ExecutionViewMgr.ConnectImage`

Call the following methods to obtain an image without connecting the image to a control:

- `ApplicationMgr.GetImageName`
- `SequenceFileViewMgr.GetImageName`
- `ExecutionViewMgr.GetImageName`

To obtain an image from an image name, you must use properties from the TestStand API, such as the `Engine.SmallImageList` property, the `Engine.LargeImageList` property, and the `Engine.Images` property.

Numeric Value Connections

A numeric value connection graphically displays a numeric value that illustrates the status of the application. For example, you can use the `ExecutionView Manager` control to connect a numeric value to a `StatusBar` pane so that the `StatusBar` pane displays a progress bar that indicates the percentage of progress made in the current execution.

The `NumericSources` enumeration defines the set of values to which you can connect. Refer to the *NI TestStand Help* for more information about each `NumericSources` enumeration constant and the manager controls to which the `NumericSources` enumeration constant applies.

Call the `ExecutionViewMgr.ConnectNumeric` method to connect a numeric source to a `StatusBar` pane. Call the `ExecutionViewMgr.GetNumericValue` method to obtain a numeric value without connecting the value to a control.

Specifying and Changing Control Connections

An application typically establishes control connections after loading the window that contains the controls to connect, but the application can establish or change control connections at any time.

You can make the same connection from a manager control to multiple visible controls. For example, if you connect two combo boxes to the sequence list of a `SequenceFileView Manager` control, both combo boxes display the selected sequence in the current file. If the selection in one combo box changes, the other combo box updates to show the new selection. However, a visible control or a connectable element of a visible control, such as a `ListBar` page or a `StatusBar` pane, can have only one connection of a particular type.

When you connect a manager control to a visible control that has an existing connection, the new connection replaces the existing connection.

Editor Versus Operator Interface Applications

An Editor application permits users to create, edit, and save sequence files. An Operator Interface application allows users only to run sequences.

Use the TestStand UI Controls to create Editor applications, Operator Interface applications, and applications that can switch between Editor Mode and Operator Mode.

Creating Editor Applications

You must enable the Editor Mode for the TestStand UI Controls to create an Editor application.

Set the `ApplicationMgr.IsEditor` property at design time to specify if Editor Mode is on or off by default. Alternatively, you can set the `ApplicationMgr.IsEditor` property in the application source code before you call the `ApplicationMgr.Start` method.

You can pass a command-line argument to override the default editing mode for the application. Pass `/editor` to set the `ApplicationMgr.IsEditor` property and pass `/operatorInterface` to clear the `ApplicationMgr.IsEditor` property. Set the `ApplicationMgr.CommandLineCanChangeEditMode` property to `False` to prevent users from changing the `ApplicationMgr.IsEditor` property from the command line.

The full-featured user interface examples allow users with sequence file editing permissions to toggle the editing mode by pressing **<Ctrl-Alt-Shift-Insert>**. To change or disable this keystroke in an application based on a full-featured example, set the `ApplicationMgr.EditModeShortcutKey` and `ApplicationMgr.EditModeShortcutModifier` properties in the designer or in the user interface source code.

License Checking

The `ApplicationMgr.Start` method verifies that a license exists to run the application. If no license exists, the `ApplicationMgr.Start` method returns an error the application displays before exiting. If an unactivated license or an evaluation license exists, the `ApplicationMgr.Start` method prompts users to activate a license.

If the `ApplicationMgr.IsEditor` property is `True`, the `ApplicationMgr.Start` method requires a license that permits editing. If you call the `ApplicationMgr.Start` method when the `ApplicationMgr.IsEditor` property is `False` and later set the `ApplicationMgr.IsEditor` property to `True`, the `ApplicationMgr.IsEditor` property returns an error if it cannot obtain a license that permits editing.

Using TestStand UI Controls in Different Environments

You can use the TestStand UI Controls in LabVIEW, LabWindows/CVI, Microsoft Visual Studio, and Visual C++.

LabVIEW

To use the TestStand UI Controls in LabVIEW, use the VIs, functions, and controls on the TestStand Functions and Controls palettes. Refer to Chapter 6, *Creating Custom User Interfaces in LabVIEW*, of the *Using LabVIEW with TestStand* manual for more information about using the TestStand UI Controls in LabVIEW.

LabWindows/CVI

To use the TestStand UI Controls in LabWindows/CVI, add the following files to the project from the `<TestStand>\API\CVI` directory:

- `tsui.fp`—ActiveX API for the TestStand UI Controls
- `tsuisupp.fp`—ActiveX API for use with less commonly used interfaces the TestStand UI Controls provide
- `tsutil.fp`—Functions that facilitate using the TestStand API and the TestStand UI Controls in LabWindows/CVI
- `tsapicvi.fp`—ActiveX API for the TestStand Engine

Include the following header files located in the `<TestStand>\API\CVI` directory in the source code files as needed:

- `tsui.h`
- `tsuisupp.h`
- `tsutil.h`
- `tsapicvi.h`

To add a TestStand UI Control to a panel in the LabWindows/CVI UIR editor, select **Create»ActiveX** and select a control that begins with TestStand UI.

Refer to Chapter 6, *Creating Custom User Interfaces in LabWindows/CVI*, of the *Using LabWindows/CVI with TestStand* manual for more information about using the TestStand UI Controls in LabWindows/CVI.

Microsoft Visual Studio

To use the TestStand UI Controls in Visual Studio, drag the TestStand UI Controls from the TestStand tab on the Visual Studio Toolbox onto a form.

When you create a new project in Visual Studio 2005, select **Project»<Project Name> Properties»Compile**, click the **Advanced Compile Options** button, and select **x86** from the **Target CPU** ring control in the Advanced Compiler Settings dialog box so the project can access the TestStand API and UI Controls on 64-bit versions of Windows.

If the Visual Studio Toolbox window does not display the TestStand tab when you edit a form or if the TestStand Interop assemblies do not appear in the Add References dialog box, exit all running copies of Visual Studio, select **Start»All Programs»National Instruments»TestStand x.x»TestStand Version Selector** to run the TestStand Version Selector utility, select the current version of TestStand, and click the **Make Active** button.

You must also add references to the TestStand Interop assemblies and the TestStand Utility (TSUtil) assembly to the project. Refer to the [Accessing the TestStand API in Visual Studio .NET 2003 and Visual Studio 2005](#) section of Chapter 5, *Module Adapters*, for more information about adding references to .NET interop assemblies for the TestStand API. Refer to the [TestStand Utility Functions Library](#) section of this chapter for more information about adding a reference to the TSUtil library for .NET.

When you create a Multiple Document Interface (MDI) application with TestStand UI Controls on an MDI child form, the Microsoft .NET Framework resets the properties you programmatically set on the TestStand UI Controls to default values when you set the `MdiParent` property on the child form. The .NET Framework resets these properties because the .NET Framework destroys and recreates ActiveX controls on a form when you set the property on the form. To avoid this issue, set the TestStand control properties after you set the `MdiParent` property on the form or place all TestStand UI Controls and other ActiveX controls on a Panel control instead of directly on the form.

Visual C++

To use the TestStand UI Controls in Visual C++, add the TSUtil Functions Library to the project as described in the [TestStand Utility Functions Library](#) section of this chapter. The `TSUtilCPP.cpp` and `TSUtilCPP.h` files automatically import the type libraries for the TestStand API and the TestStand UI Controls.

You can view the header files the `#import` directive generates for the TestStand API type libraries by opening the `tsui.tlh`, `tsuisupp.tlh`, and `tsapi.tlh` files Visual C++ creates in the `Debug` or `Release` directory. The header files the `#import` directive generates define a C++ class for each object class in the TestStand API. The `T` prefix in class names denotes ActiveX controls and objects you can create without calling another class. The header files use macros to define a corresponding smart pointer class for each object class. Each smart pointer class uses the name of its corresponding class and adds a `Ptr` suffix. Typically, you use only smart pointer classes in an application because the smart pointer releases the reference to the object when the pointer is destroyed. For example, instead of using the `SequenceFile` class, use the `SequenceFilePtr` class.



Note National Instruments recommends, in accordance with Microsoft guidelines, using the classes the `#import` directive generates to call the TestStand ActiveX API instead of using the Class Wizard tool to generate MFC wrapper class files.

Select **Insert ActiveX Control** from the dialog box context menu and select a control that begins with `TestStand UI` to add a TestStand UI Control to a dialog box as a resource.



Note If you programmatically create a TestStand UI Control in an MFC container, you must remove the `WS_CLIPSIBLINGS` style from the control window for the TestStand UI Control to remain visible inside an MFC Group Box control. If you do not remove the `WS_CLIPSIBLINGS` style, a native MFC control always obscures the TestStand UI Control, even if the MFC control comes after the TestStand UI Control in the tab order.

Obtaining an Interface Pointer and CWnd for an ActiveX Control

Complete the following steps to obtain an interface pointer to an ActiveX control, such as a TestStand UI control, that you insert into an MFC dialog resource.

Using GetDlgItem

1. Add a CWnd member to the dialog class for the control as follows:
CWnd mExprEditCWnd;
2. Insert the following code into the OnInitDialog method of the dialog class:

```
mExprEditCWnd.Attach(GetDlgItem
(IDC_MYEXPRESSIONEDIT)->m_hWnd);
```

3. Obtain the interface pointer from the CWnd member as follows:

```
TSUI::IExpressionEditPtr myExprEdit =
mExprEditCWnd.GetControlUnknown();
```



Note National Instruments does not recommend using DoDataExchange to obtain an interface pointer and CWnd for a TestStand ActiveX User Interface Control because the pointer can be invalid in some instances. Use DoDataExchange only when controls are windowless or do not recreate internal windows.

Handling Events

TestStand UI Controls generate events to notify an application of user input and of application events, such as an execution completing. The visible controls generate user input events, such as `KeyDown` or `MouseDown`. The manager controls generate application state events, such as the `ApplicationMgr.SequenceFileOpened` event or the `ApplicationMgr.UserChanged` event. You can handle events according to the needs of the application, as shown in Table 9-3.

Table 9-3. Creating Event Handlers in Specific ADEs

ADE	Description
LabVIEW	Register event handler VIs with the Register Event Callback function. Refer to the <i>Handling Events</i> section of Chapter 6, <i>Creating Custom User Interfaces in LabVIEW</i> , of the <i>Using LabVIEW with TestStand</i> manual for more information about handling events from the TestStand UI Controls in LabVIEW.
LabWindows/CVI	Install ActiveX event callback functions by calling the <code>TSUI_<object class>EventsRegOn<event name></code> functions in <code>tsui.fp</code> . Refer to the <i>Handling Events</i> section of Chapter 6, <i>Creating Custom User Interfaces in LabWindows/CVI</i> , of the <i>Using LabWindows/CVI with TestStand</i> manual for more information about handling events from the TestStand UI Controls in LabWindows/CVI.
.NET	Create .NET control event handlers from the form designer.
C++ (MFC)	Create ActiveX event handlers from the Message Maps page in the Class Wizard dialog box.

Events Typical Applications Handle

When you create an application, you can direct the application to handle any subset of the available TestStand UI Control events. However, an application typically handles the `ExitApplication`, `Wait`, `ReportError`, `DisplaySequenceFile`, and `DisplayExecution` events.

ExitApplication

The Application Manager control generates this event to request that the application exit. Handle this event by directing the application to exit normally. Refer to the [Startup and Shutdown](#) section of this chapter for more information about shutting down the application.

Wait

The Application Manager control generates this event to request that the application display or remove a busy indicator. Handle this event by displaying or removing a wait cursor according to the value of the `showWait` event parameter.

ReportError

The Application Manager control generates this event to request that the user interface return an error during user input or during an asynchronous operation. Handle this event by displaying the error code and description in a dialog box or by appending the error code and description to an error log. The `ApplicationMgr.ReportError` event indicates an application error, not a sequence execution error. The `ApplicationMgr.BreakOnRunTimeError` event indicates a sequence execution error.

DisplaySequenceFile

The Application Manager control generates this event to request that the application display a particular sequence file. Handle this event by displaying the sequence file by setting the `SequenceFileViewMgr.SequenceFile` property. If the application has only a single window, set this property on the `SequenceFileView Manager` control that resides on the window. If the application displays each sequence file in a separate window using separate `SequenceFileView Manager` controls, call the `ApplicationMgr.GetSequenceFileViewMgr` method to find the `SequenceFileView Manager` control that currently displays the sequence file so you can activate the window that contains the sequence file. If no `SequenceFileView Manager` control currently displays the sequence file, a multiple window application can create a new window that contains a `SequenceFileView Manager` control. The application can then set the `SequenceFileViewMgr.SequenceFile` property to display the sequence file in the new window.

DisplayExecution

The Application Manager control generates this event to request that the application display a particular execution. Handle this event by displaying the execution by setting the `ExecutionViewMgr.Execution` property. If the application has only a single window, set this property on the `ExecutionView Manager` control that resides on the window. If the application displays each execution in a separate window using separate `ExecutionView Manager` controls, call the `ApplicationMgr.GetExecutionViewMgr` method to find the `ExecutionView Manager` control that currently displays the execution so you can activate the window that contains the execution. If no `ExecutionView Manager` control currently displays the execution, a multiple window application can create a new window that contains an

ExecutionView Manager control. The application can then set the `ExecutionViewMgr.Execution` property to display the execution in the new window.

Startup and Shutdown

As a final step in the initialization of the application, call the `ApplicationMgr.Start` method to initialize the Application Manager control and launch the LoginLogout Front-End callback if you did not set the `ApplicationMgr.LoginOnStart` property to `False`.

Complete the following steps to shut down the application.

1. If the application holds any references to TestStand objects, such as sequence files or executions, handle the `ApplicationMgr.QueryShutDown` event by canceling the shutdown process or releasing the TestStand object references the application holds.
2. Call the `ApplicationMgr.ShutDown` method. If the method returns `True`, exit the application. If the method returns `False`, do not exit the application. Leaving the application running allows the method to shut down any running executions and unload sequence files. If the shut down process completes, the Application Manager control generates the `ApplicationMgr.ExitApplication` event to notify you to exit the application. If the application cancels the shutdown process, the Application Manager control generates the `ApplicationMgr.ShutDownCancelled` event, which occurs when users choose not to terminate a busy execution.



Note When you use the TestStand UI Controls to create an Exit button or an Exit menu item that invokes the `Exit` command, the button or menu item automatically calls the `ApplicationMgr.ShutDown` method for you.

3. Exit the application in the event handler you create for the `ApplicationMgr.ExitApplication` event. The window in which the Application Manager control resides must exist until you receive the `ApplicationMgr.ExitApplication` event.

TestStand Utility Functions Library

Use the TSUtil Functions Library to use certain aspects of the TestStand API in particular ADEs. Many TSUtil functions operate on environment-specific objects, such as menus, that the environment-neutral TestStand API cannot access. The functions available in TSUtil vary according to the ADE.

The TSUtil library contains functions to insert menu items that automatically execute commands the TestStand UI Controls API provides. The TSUtil library also provides functions to help localize the strings on a user interface.

Refer to the [Menus and Menu Items](#) section of this chapter for more information about using TSUtil functions to create menu items that perform common TestStand commands. Refer to the [Localization](#) section of this chapter for more information about displaying application user interface strings in a different language.

Table 9-4 describes how to use the TSUtil library in different ADEs. If Table 9-4 does not include the ADE you use, a version of TSUtil does not exist for the ADE.

Table 9-4. Using the TSUtil Library in Different ADEs

ADE	Help Location	Files	How to Use
LabVIEW	Context help for each VI and in the <i>NI TestStand VIs and Functions Help</i> , accessible by right-clicking the VI and selecting Help from the shortcut menu or by selecting Help»NI TestStand VIs and Functions	VIs on the Functions»TestStand palette _TSUtility.llb located in <TestStand>\API\LabVIEW	Place VIs on the block diagram. Refer to Chapter 6, <i>Creating Custom User Interfaces in LabVIEW</i> , of the <i>Using LabVIEW with TestStand</i> manual for more information about using the TSUtil library in LabVIEW.
LabWindows/CVI	Function panels (TSUtil.fp)	TSUtil.c, TSUtil.h, TSUtil.fp, and TSUtil.obj located in <TestStand>\API\CVI	Insert TSUtil.fp into the LabWindows/CVI project. Include TSUtil.h in the source files as needed. The names of TestStand-related functions begin with a TS_ prefix. Refer to Chapter 6, <i>Creation Custom User Interfaces in LabWindows/CVI</i> , of the <i>Using LabWindows/CVI with TestStand</i> manual for more information about using the TSUtil library in LabWindows/CVI.
.NET Languages	In the Object Browser and in the source window using Intellisense	National Instruments. TestStand.Utility.dll located in <TestStand>\API\DotNet\Assemblies\CurrentVersion	Add a reference to the assembly to the project. The classes in this assembly reside in the National Instruments.TestStand.Utility namespace. Refer to the Adding Assembly References in Visual Studio section of this chapter for more information about adding references to assembly files in Visual Studio.
C++ (MFC)	Comments in the C++ header file, TSUtilCPP.h	TSUtilCPP.cpp and TSUtilCPP.h located in <TestStand>\API\VC	Add TSUtilCPP.cpp to the project once. Include TSUtilCPP.h in the source files as needed. The classes in this library reside in the TSUtil namespace.

You can use the source code for one of the existing TSUtil libraries as a guide to write your own code that performs similar functionality.

Adding Assembly References in Visual Studio

Complete the following steps to add an assembly reference in Visual Studio 2005.

1. Select the project in the Solution Explorer.
2. Select **Project»Add Reference** to launch the Add Reference dialog box.
3. Click the **.NET** tab and select `National Instruments.TestStand.Utility` from the list of components.
4. Click **OK** to close the Add Reference dialog box.

Complete the following steps to add an assembly reference in Visual Studio .NET 2003. You must use an assembly compatible with TestStand 3.5 or earlier because Visual Studio .NET 2003 requires the assembly to be compatible with the .NET Framework 1.1, and the assemblies in TestStand 4.0 and later require the .NET Framework 2.0.

1. Select the project in the Solution Explorer.
2. Select **Project»Add Reference** to launch the Add Reference dialog box.
3. Click the **.NET** tab and click the **File Browse** button to launch the Select Components dialog box.
4. Navigate to the `<TestStand>\API\DotNet\Assemblies\PreviousVersion\3.5` directory.
5. Select `National Instruments.TestStand.Utility.dll` and click **Open**.
6. Click **OK** to close the Add Reference dialog box.

Menus and Menu Items

TestStand applications that provide non-trivial menus can require a large amount of source code to build and update the state of menus and to handle events for menu items. Use the TSUtil functions to create menu items that invoke TestStand commands to greatly reduce the amount of code required to implement menus in an application. TestStand automatically dims or enables these menu items according to the application state and sets their captions according to the language selection. The menu items execute commands automatically so that the application does not need to handle menu events or provide command implementations.

The application can also insert sets of dynamic menu items, such as a set of menu items to open files from the most recently used file list or a set of menu items that run the current sequence with each available Process Model entry point. To create TestStand menu items, you must first add TSUtil to the project as described in the [TestStand Utility Functions Library](#) section of this chapter.



Note The TSUtil .NET menu functions support using the MainMenu control in Visual Studio .NET 2003 but do not support using the MenuStrip control in Visual Studio 2005. To access the .NET MainMenu control in the Visual Studio 2005 Toolbox, select **Choose Items** from the context menu on the Toolbox pane, enable **MainMenu 2.0** on the .NET Framework Components tab of the Choose Toolbox Items dialog box, and click **OK** to close the dialog box. The full-featured .NET example user interface applications use MainMenu to display menus.

Updating Menus

The contents of a menu can vary depending on the current selection, other user input, or asynchronous execution state changes. Instead of updating a menu in response to any event or change that might affect the menu, update the state of a menu just before the menu displays when the user opens the menu. Table 9-5 lists the notification method different ADEs use to notify the application when a user is about to open a menu.

Table 9-5. Menu Open Notification Methods in Different ADEs

ADE	Menu Open Notification Method
LabVIEW	<p><This VI>:Menu Activation? event</p> <p>Refer to the <i>Menu Bars and Menu Event Handling</i> section of Chapter 6, <i>Creating Custom User Interfaces in LabVIEW</i>, of the <i>Using LabVIEW with TestStand</i> manual for more information about determining when a menu is about to open in LabVIEW.</p>
LabWindows/CVI	<p>InstallMenuDimmerCallback</p> <p>Refer to the <i>Menu Bars</i> section of Chapter 6, <i>Creating Custom User Interfaces in LabWindows/CVI</i>, of the <i>Using LabWindows/CVI with TestStand</i> manual for more information about determining when a menu is about to open in LabWindows/CVI.</p>

Table 9-5. Menu Open Notification Methods in Different ADEs (Continued)

ADE	Menu Open Notification Method
.NET	Form.MenuStart
C++ (MFC)	CWnd::OnInitMenuPopup

Use the `RemoveMenuCommands`, `InsertCommandsInMenu`, and `CleanupMenu` TSUtil functions to handle the menu open notifications and remove and reinsert TestStand menu items. You can remove and insert TestStand commands in menus that contain non-TestStand menu items.

The `InsertCommandsInMenu` function accepts an array of `CommandKinds` enumeration constants. Depending on the element value and the application state, each array element can create a single menu item, a set of several menu items, or no menu items. The `CommandKinds` enumeration also provides constants that expand into the full set of items commonly found in test application top-level menus, such as the File menu, Debug menu, or Configure menu.

Refer to the *TestStand Utility Functions Library* section of this chapter for more information about the utility functions. Refer to the examples in the `<TestStand>\UserInterfaces\Full-Featured` directory for sample code that handles menu open notification events.

Localization

The `StationOptions.Language` property specifies the current language. Localized TestStand applications use the `Engine.GetResourceString` method to obtain text in the current system language from language resource files. Refer to the *Creating String Resource Files* section of Chapter 8, *Customizing and Configuring TestStand*, for more information about creating string resource files.

Call the `ApplicationMgr.LocalizeAllControls` method to localize all the user-visible TestStand UI Control strings you configure at design time. Using the `ApplicationMgr.LocalizeAllControls` method reduces the number of strings you must explicitly localize using the `Engine.GetResourceString` method by localizing items such as list column headers in the `SequenceView` control, text in the `StatusBar` pane, captions in the `Button` control, and captions in the `ListBar` page.

Buttons and menu items you connect to commands automatically localize caption text. Refer to the [Command Connections](#) section of this chapter for more information about connecting buttons and menu items to commands.

The `ApplicationMgr.LocalizeAllControls` method operates only on TestStand UI Controls. For other controls and user interface elements, the application must set each item of localized text. Table 9-6 lists the TSUtil library functions you can use to localize non-TestStand controls and menu items.

Table 9-6. TSUtil Library Localization Functions in Different ADEs

ADE	TSUtil Library Localization Function
LabVIEW	TestStand - Localize Front Panel.vi TestStand - Localize Menu.vi TestStand - Get Resource String.vi
LabWindows/CVI	TS_LoadPanelResourceStrings TS_LoadMenuBarResourceStrings TS_SetAttrFromResourceString TS_GetResourceString
.NET	Localizer.LocalizeForm Localizer.LocalizeMenu
C++ (MFC)	Localizer.LocalizeWindow Localizer.LocalizeMenu Localizer.LocalizeString

Refer to the [TestStand Utility Functions Library](#) section of this chapter for more information about the TSUtil library.

User Interface Application Styles

Although you can use the TestStand UI Controls to create any type of application, the single window, multiple window, and no visible window formats are the most common. Applications of a particular style usually share a similar implementation strategy, particularly with respect to the use of the TestStand manager controls.

Single Window

A single window application typically displays one execution and sequence file at a time. Users can select the execution and sequence file to display from a `ListBar`, `ComboBox`, or `ListBox` control. The examples in the `<TestStand>\UserInterfaces\Full-Featured` and `<TestStand>\UserInterfaces\Simple` directories are single window applications.

A single window application contains one `Application Manager` control, one `SequenceFileView Manager` control, and one `ExecutionView Manager` control. To display sequences, connect the `SequenceFileView Manager` and `ExecutionView Manager` controls to separate `SequenceView` controls, alternate a connection from each manager control to a single `SequenceView` control, or leave one or both manager controls unconnected to a `SequenceView` control.

In the examples in the `Full-Featured` directory, the `SequenceFileView Manager` control and the `ExecutionView Manager` control connect to separate `SequenceView` controls, and only one `SequenceView` control is visible at a time. Visibility depends on if you select to view sequence files or executions.

In the examples in the `Simple` directory, the `ExecutionView Manager` control connects to the `SequenceView` control. Because the `SequenceFileView Manager` control does not connect to a `SequenceView` control, these examples display only sequences for the current execution, not sequences from the sequence file selection.

Multiple Window

A multiple window application includes at least one window that always exists to contain the `Application Manager` control. Although this window can be visible or invisible, it is typically visible and contains controls for users to open sequence files.

For each sequence file users open, the application creates a `Sequence File` window that contains a `SequenceFileView Manager` control and a `SequenceView` control to which the manager control connects. The application sets the `SequenceFileViewMgr.UserData` property to attach a handle, reference, or pointer that represents the window. When the application receives the `ApplicationMgr.DisplaySequenceFile` event, the application calls `ApplicationMgr.GetSequenceFileViewMgr` to determine if a `SequenceFileView Manager` control currently displays the sequence

file. If so, the application retrieves the window from the `SequenceFileViewMgr.UserData` property and activates the window. If no window currently displays the sequence file, the application creates a new window and sets the `SequenceFileViewMgr.SequenceFile` property to display the sequence file. Because the window displays only this sequence file, the application also sets the `SequenceFileViewMgr.ReplaceSequenceFileOnClose` property to `False`.

If a Sequence File window attempts to close and the `SequenceFileViewMgr.SequenceFile` property is `NULL`, the application closes the window immediately. If the `SequenceFileViewMgr.SequenceFile` property is not `NULL`, the application does not close the window. Instead, the application passes the sequence file to the `ApplicationMgr.CloseSequenceFile` method. When the application receives the `SequenceFileViewMgr.SequenceFileChanged` event with a `NULL` sequence file event parameter, the application closes the window that holds the SequenceFileView Manager control.

The Sequence File window contains controls for users to execute the sequence file the window displays. For each execution users start, the application creates an Execution window that contains an ExecutionView Manager control and a SequenceView control to which the manager control connects. The application sets the `ExecutionViewMgr.UserData` property to attach a handle, reference, or pointer that represents the window. When the application receives the `ApplicationMgr.DisplayExecution` event, the application calls the `ApplicationMgr.GetExecutionViewMgr` method to determine if an ExecutionView Manager control currently displays the execution. If so, the application retrieves the window from the `ExecutionViewMgr.UserData` property and activates the window. If no window currently displays the execution, the application creates a new window and sets the `ExecutionViewMgr.Execution` property to display the execution. Because the window displays only this execution, the application also sets the `ExecutionViewMgr.ReplaceExecutionOnClose` property to `False`.

If an Execution window attempts to close and the `ExecutionViewMgr.Execution` property is `NULL`, the application closes the window immediately. If the `ExecutionViewMgr.Execution` property is not `NULL`, the application does not close the window. Instead, the application passes the execution to the `ApplicationMgr.CloseExecution` method. The application does not

immediately close the Execution window to ensure that the window exists until the execution the window displays completes. When the application receives the `ExecutionViewMgr.ExecutionChanged` event with a `NULL` execution event parameter, the application closes the window that holds the ExecutionView Manager control.

A multiple window application can display multiple child windows instead of displaying sequence files and executions in separate top-level windows. Child windows can be visible or reside on tab control pages or similar locations that allow users to easily select which child window to view.

No Visible Window

An application without a visible window is similar to a single window application. The application can execute command-line arguments and then exit, or the application can have a different mechanism to determine which files to load and execute. Although an invisible application does not require an ExecutionView Manager control, the application can use a SequenceFileView Manager control to provide methods to launch an execution for a sequence file. Use the `SequenceFileViewMgr.ExecutionEntryPoints` property, the `SequenceFileViewMgr.Run` method, the `SequenceFileViewMgr.RunSelectedSteps` method, the `SequenceFileViewMgr.LoopOnSelectedSteps` method, and the `SequenceFileViewMgr.GetCommand` method to launch executions in an application without a visible window.

Command-Line Arguments

The Application Manager control automatically processes the command-line argument that invokes the application when you call the `ApplicationMgr.Start` method. Set the `ApplicationMgr.ProcessCommandLine` property to `False` before you call the `ApplicationMgr.Start` method to disable command-line processing. Refer to the [Configuring Sequence Editor and User Interface Startup Options](#) section of Chapter 8, [Customizing and Configuring TestStand](#), for a description of the command-line arguments the Application Manager control processes.

You can also handle the

`ApplicationMgr.ProcessUserCommandLineArguments` event to support additional command-line arguments.

The `ApplicationMgr.ProcessUserCommandLineArguments` event occurs when the Application Manager control parses and processes an

unrecognized command-line flag. Refer to the *NI TestStand Help* for more information about using the

`ApplicationMgr.ProcessUserCommandLineArguments` event to support user command-line flags in a user interface.

Persistence of Application Settings

The TestStand Engine stores Station Options dialog box settings and other settings that apply to all TestStand applications. However, each user interface also stores additional custom settings, including breaking on the first step of execution, breaking when a step fails, and listing the most recently used sequence files. The Application Manager control stores these settings in the configuration file the `ApplicationMgr.ConfigFilePath` property specifies.

The `ApplicationMgr.BreakonFirstStep`, `ApplicationMgr.PromptForOverwrite`, `ApplicationMgr.EditReadOnlyFiles`, `ApplicationMgr.MakeStepNamesUnique`, and `ApplicationMgr.SaveOnClose` properties persist to the configuration file. Setting the value of one of these properties on the Application Manager control in a designer sets the default value for the property. The Application Manager control stores the default value in the configuration file it creates if a configuration file does not already exist. If the configuration file already exists, the Application Manager control loads the values of these properties from the file.

Configuration File Location

The default value of the `ApplicationMgr.ConfigFilePath` property is `%TestStandLocalAppData%\UserInterface.xml`, in which `%TestStandLocalAppData%` is a macro that expands to a directory to which the currently logged-in user has permission to write files. The directory is typically `<User Directory>\Local Settings\ Application Data\National Instruments\TestStand x.x` on Windows 2000/XP and `<User Directory>\AppData\Local\ National Instruments\TestStand x.x` on Windows Vista. Set the `ApplicationMgr.ConfigFilePath` property before the application calls the `ApplicationMgr.Start` method to change the configuration file location.

If you specify a relative file path or just a filename, the file location is relative to the directory that contains the application. If users who do not have Windows administrator privileges can run the application, you must store the configuration file in a location to which users have permission to write files.

Adding Custom Application Settings

After the application calls the `ApplicationMgr.Start` method, complete the following steps to add your own setting to persist in the configuration file.

1. Access the `ApplicationMgr.ConfigFile` property to obtain the `PropertyObjectFile` that holds the contents of the configuration file.
2. Access the `PropertyObjectFile.Data` property to obtain the `PropertyObject` that holds the application settings.
3. Ensure your custom setting exists in the `PropertyObject` by setting a default value of the setting by calling a method, such as the `PropertyObject.SetValBoolean` method, with a lookup string, such as “`CustomSettings.MyExampleBooleanSetting`,” and an options parameter of `PropOption_SetOnlyIfDoesNotExist`.
4. Call a method, such as the `PropertyObject.GetValBoolean` method, to obtain the current value of the custom option.
5. Call a method, such as the `PropertyObject.SetValBoolean` method, with an options parameter of `PropOption_NoOptions` to set the custom option in response to user input.

When you call the `ApplicationMgr.ShutDown` method or change any Application Manager control setting, the Application Manager control persists the application settings to the configuration file. You can also call the `PropertyObjectFile.WriteFile` method at any time to persist the settings.

Documenting Custom User Interfaces

You can use the *Using the TestStand User Interfaces* document, located at `<TestStand>\Doc\UsingtheTestStandUserInterfaces.doc`, as a starting point for creating a custom manual for user interface applications you customize based on the TestStand full-featured user interface examples. In addition, the menus for the TestStand full-featured user interface examples include `CommandKind_DefaultHelpMenu_Set`, which

contains a set of commands that corresponds to the typical items in the Help menu of a TestStand application, including support for using the <F1> key to display help for the currently active TestStand UI control.

Deploying a User Interface

Refer to the *Distributing a User Interface* section of Chapter 14, *Deploying TestStand Systems*, for more information about deploying a TestStand User Interface application.

Authenticode Signatures for Windows Vista

Authenticode signatures can help identify the publisher of a binary file and can help ensure that a binary file has not been modified since publication. Refer to Microsoft documentation for more information about Authenticode signatures.

Add an Authenticode signature to a TestStand user interface you create if you plan to allow users to download the user interface from a non-trusted public site and you want the operating system to identify your company as the publisher of the user interface. Also add an Authenticode signature to a user interface you create if the user interface requires administrator privileges to run on Windows Vista and you want the UAC elevation prompt to identify your company as the publisher of the user interface.

To verify an Authenticode signature, the requesting computer must connect to the Internet to obtain a current Certificate Revocation List (CRL). For .NET applications, the .NET Common Language Runtime (CLR) verifies Authenticode signatures for assemblies. If the computer that loads the assembly is not connected to the Internet, the CLR waits 15 seconds before timing out.

Complete the following steps to disable CRL validation in Microsoft Internet Explorer to avoid the timeout period on the computer, even if the default browser on the computer is not Internet Explorer. Using the Internet Explorer Internet Options to disable CRL validation does not expose the computer to any additional security threats.

1. Navigate to **Internet Options** on the Windows Control Panel and click the Advanced tab.
2. In the Security section, disable the **Check for publisher's certificate revocation** option.

Alternatively, you can disable CRL validation by setting the registry key value of HKCU\Software\Microsoft\Windows\CurrentVersion\WinTrust\Trust Providers\Software Publishing\State to 0x00023e00. To enable CRL validation, set the registry key value to 0x00023c00.

When you disable CRL validation to avoid the timeout period, the CLR does not validate Authenticode-signed assemblies and does not grant the assemblies publisher evidence or publisher identity permissions, which is the same result when a timeout occurs. If the assemblies need these permissions, the computer must connect to the Internet or you must download a current CRL every 10–15 days.

As an alternative to disabling CRL validation for the entire computer, you can work around CRL validation if an application that uses the .NET Framework 2.0 and that has an Authenticode signature experiences the 15-second load time delay. Microsoft provides a fix you can download so you can correct this delay for .NET Framework 2.0 applications. The .NET Framework 2.0 Service Pack 1 also includes this fix. Refer to Microsoft Knowledge Base article 936707 at support.microsoft.com/kb/936707 for more information about correcting delays in .NET Framework 2.0 applications that use Authenticode signatures.

The TestStand Sequence Editor and user interface examples do not include Authenticode signatures because National Instruments distributes TestStand through trusted channels and because the TestStand Sequence Editor and user interface examples do not require administrator privileges to run on Windows Vista. Additionally, National Instruments finds the 15-second load time delay on isolated networks unacceptable and believes that you should use discretion when disabling CRL validation. Therefore, if you run the sequence editor or example user interfaces as administrator on Windows Vista, the UAC elevation prompt does not identify the sequence editor or example user interface as a National Instruments product.

Application Manifests

When an application launches on Windows Vista, the User Account Control (UAC) security component determines whether to grant the application administrative privileges. A user that logs into Windows Vista as a standard user can write only to specific locations on disk and in the registry. Standard user is the default login for Windows Vista.

Microsoft recommends that applications run without requiring administrator privileges. If you design applications that do not attempt to access protected areas of the operating system, all users can run the application as intended without requiring administrator privileges. You can also include manifests to specify the execution level the application requires.

If an application does not specify an execution level in its manifest, the UAC launches the application with the standard or administrator privileges of the user. With standard privileges, the system uses virtualization to redirect any read and write operations for system files and registry keys to a per-user location instead of the actual system copy of the file or registry key. Do not create applications that rely on virtualization to perform these types of administrative operations.

The default TestStand user interface application binary files include manifests that instruct the UAC to execute the application without virtualization and without requiring administrative privileges. LabVIEW 8.5 and later automatically include a default manifest in built applications. LabWindows/CVI 8.5 and later allow you to specify a manifest for built applications. When you build the application, refer to the documentation for the ADE you used for more information on how to include a manifest.

Customizing Process Models and Callbacks

You can customize the default TestStand process models and callbacks. Customize callbacks to implement custom functionality specific to UUT models. Customize process models to implement functionality that is standard throughout an organization and applies to all or most UUTs. Use callbacks to implement functionality you might change. Use process models to implement functionality you are unlikely to change. Refer to Appendix A, *Process Model Architecture*, for more information about the TestStand process models.

Modifying Process Model Sequence Files

You must modify the process model sequence files directly to make changes that apply wherever TestStand uses the process model.

TestStand installs the `SequentialModel.seq`, `ParallelModel.seq`, and `BatchModel.seq` process model sequence files and supporting files in the `<TestStand>\Components\Models\TestStandModels` directory. To modify the installed process model files or to create a new process model file, copy the `<TestStand>\Components\Models\TestStandModels` directory to the `<TestStand Public>\Components\Models` directory and make changes to the copies of the files. When you copy installed files to modify, 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.

In addition to editing process model sequence files, you can convert sequence files to process model sequence files. Complete the following steps to specify a sequence file as a process model sequence file.

1. Select the sequence file and select **Edit»Sequence File Properties**.
2. In the Sequence File Properties dialog box, click the **Advanced** tab.
3. Select **Model** from the Type ring control.
4. Click **OK**.

Although you edit a process model sequence file in a regular Sequence File window, the file includes Model entry points and Model callbacks. TestStand maintains special properties for entry point and callback sequences, and you can specify the values of these properties when you edit the sequences in a process model file.

When you access the Sequence Properties dialog box for any sequence in a model file, the dialog box contains a Model tab you use to specify if the sequence is a normal sequence, a callback sequence, or an entry point sequence.

Normal Sequences

A normal sequence is any sequence other than a callback or an entry point. In process model files, use normal sequences as Utility subsequences that entry points or callbacks call. When you select **Normal** from the Type ring control on the Model tab of the Sequence Properties dialog box, the Model tab does not include any other options.

Callback Sequences

Model callbacks are sequences entry point sequences call and client sequence files can override. Use Model callbacks to customize the behavior of a process model for each client sequence file that uses the process model. By defining one or more Model callbacks in a process model file, you specify the set of process model operations you can customize from a client sequence file.

Complete the following steps to define a Model callback.

1. Add a sequence to the process model file.
2. Select **Edit»Sequence Properties** to launch the Sequence Properties dialog box.
3. Click the **Model** tab and select **Callback** from the Type ring control.

4. Click **OK**.
5. Call the new sequence you just created from the process model.

You can override a callback in the process model sequence file by using the Sequence File Callbacks dialog box to create a sequence with the same name but different functionality in the client sequence file. Select **Edit» Sequence File Callbacks** to launch the Sequence File Callbacks dialog box. Refer to the *NI TestStand Help* for more information about the Sequence File Callbacks dialog box.

Some Model callbacks, such as the TestReport callback in the default process model, are sufficient for handling specific types of operations. Other Model callbacks are placeholders you override with sequences in the client sequence file. For example, the MainSequence callback in the default process model file is a placeholder for the MainSequence callback you create in the client sequence file.

A primary process model file can directly call model callback sequences in a secondary process model file. At run time, if the client sequence file of the primary sequence file implements a callback defined in the secondary process model file, TestStand invokes the callback sequence in the client sequence file, even if the primary process model file does not define the callback. You must add a copy of the callback sequence to the primary model file for the callback to appear in the Sequence File Callbacks dialog box for the client sequence file.

Entry Point Sequences

You can invoke Execution entry point sequences and Configuration entry point sequences from the TestStand Sequence Editor or user interface menus to run client files or to configure model settings.

Execution entry points run test programs typically by calling the MainSequence callback in the client sequence file. The TestStand Sequential, Parallel, and Batch process models contain the following Execution entry points:

- **Test UUTs**—Tests and identifies multiple UUTs or batches of UUTs in a loop.
- **Single Pass**—Tests one UUT or a single batch of UUTs without identifying the UUTs.

By default, the Execute menu lists Execution entry points only when the active window contains a sequence file that uses the process model.

Configuration entry points configure a feature of the process model and usually save the configuration information in a .ini file the <TestStand Application Data>\Cfg directory. The TestStand process models contain the following Configuration entry points:

- **Report Options**—Launches the Report Options dialog box, in which you enable UUT report generation and configure the report type and contents of the report files.
- **Database Options**—Launches the Database Options dialog box, in which you enable UUT result logging and configure the schema for mapping TestStand results to database tables and columns.
- **Model Options**—Launches the Model Options dialog box, in which you configure the number of test sockets and other process model-related options.

By default, the Configure menu lists the Configuration entry points.

Modifying Callbacks

TestStand includes Engine callback and Front-End callback sequences you can customize to meet specific needs.

Engine Callbacks

The TestStand Engine invokes a set of Engine callbacks at specific points during execution. TestStand defines the set of Engine callbacks and the callback names because the TestStand Engine controls the execution of steps and the loading and unloading of sequence files.

Use Engine callbacks to configure TestStand to call certain sequences at various points during a test, including before and after the execution of individual steps, before and after interactive executions, after loading a sequence file, or before unloading a sequence file.

TestStand categorizes Engine callbacks according to the file in which the callback sequence appears. You can define Engine callbacks in sequence files, process model files, and in `StationCallbacks.seq` in the <TestStand Public>\Components\Callbacks\Station directory.



Note TestStand does not predefine any Station Engine callbacks in `StationCallbacks.seq` in the <TestStand>\Components\Callbacks\Station directory but might in a future version of TestStand.

TestStand invokes Engine callbacks in normal sequence files only when executing steps in the sequence file or when loading or unloading the sequence file. TestStand invokes Engine callbacks in process model files when executing steps in the model file, steps in sequences the model calls, and steps in any nested calls to subsequences. TestStand invokes Engine callbacks in `StationCallbacks.seq` when TestStand executes steps on the test station.

Table 10-1 lists the engine callbacks TestStand defines, indicates where you must define the callback sequence, and specifies when the engine calls the callback.

Table 10-1. Engine Callbacks

Engine Callback	Where You Define the Callback	When the Engine Calls the Callback
SequenceFilePreStep	Any sequence file	Before the engine executes each step in the sequence file.
SequenceFilePostStep	Any sequence file	After the engine executes each step in the sequence file.
SequenceFilePreInteractive	Any sequence file	Before the engine begins an interactive execution of steps in the sequence file.
SequenceFilePostInteractive	Any sequence file	After the engine completes an interactive execution of steps in the sequence file.
SequenceFileLoad	Any sequence file	When the engine loads the sequence file into memory.
SequenceFileUnload	Any sequence file	When the engine unloads the sequence file from memory.
SequenceFilePostResultListEntry	Any sequence file	After the engine fills out the step result for a step in the sequence file.
SequenceFilePostStepRuntimeError	Any sequence file	After a step in the sequence file generates a run-time error.

Table 10-1. Engine Callbacks (Continued)

Engine Callback	Where You Define the Callback	When the Engine Calls the Callback
SequenceFilePostStepFailure	Any sequence file	After a step in the sequence fails.
ProcessModelPreStep	Process model file	Before the engine executes each step in any client sequence file the process model calls and each step in any resulting subsequence calls.
ProcessModelPostStep	Process model file	After the engine executes each step in any client sequence file the process model calls and each step in any resulting subsequence calls.
ProcessModelPreInteractive	Process model file	Before the engine begins an interactive execution of steps in a client sequence file and steps in any resulting subsequence calls.
ProcessModelPostInteractive	Process model file	After the engine completes an interactive execution of steps in a client sequence file and steps in any resulting subsequence calls.
ProcessModelPostResultListEntry	Process model file	After the engine fills out the step result for a step in any client sequence file the process model calls or in any resulting subsequence calls.

Table 10-1. Engine Callbacks (Continued)

Engine Callback	Where You Define the Callback	When the Engine Calls the Callback
ProcessModelPostStepRuntimeError	Process model file	After a step generates a run-time error when the step is in a client sequence file the process model calls or in any resulting subsequence calls.
ProcessModelPostStepFailure	Process model file	After a step fails when the step is in a client sequence file the process model calls or in any resulting subsequence calls.
StationPreStep	StationCallbacks.seq	Before the engine executes each step in any sequence file.
StationPostStep	StationCallbacks.seq	After the engine executes each step in any sequence file.
StationPreInteractive	StationCallbacks.seq	Before the engine begins any interactive execution.
StationPostInteractive	StationCallbacks.seq	After the engine completes any interactive execution.
StationPostResultListEntry	StationCallbacks.seq	After the engine fills out the step result for a step in any sequence file.
StationPostStepRuntimeError	StationCallbacks.seq	After any step generates a run-time error.
StationPostStepFailure	StationCallbacks.seq	After any step fails.

You can use Engine callbacks in the following ways:

- Use the `SequenceFileLoad` callback to ensure that you configure external resources the sequence file uses only once before you execute the sequence. Usually, you initialize devices a sequence requires by creating steps in the Setup step group for the sequence. However, if you call the sequence repeatedly, you can move the Setup steps into a `SequenceFileLoad` callback for the subsequence file so that the steps run only when the sequence file loads.
- Use the `StationPreStep` and `StationPostStep` callbacks to accumulate statistics on all steps that execute on the test station. You can inspect the name and types of steps that accumulate data on specific steps.

Caveats for Using Engine Callbacks

Consider the following issues when you define Engine callbacks:

- If you define a `SequenceFilePreStep`, `SequenceFilePostStep`, `SequenceFilePreInteractive`, or `SequenceFilePostInteractive` callback in a process model file, the callback applies only to the steps in the process model file.
- Do not define a `SequenceFileLoad` or `SequenceFileUnload` callback in the `StationCallbacks.seq` because TestStand does not call these callbacks.
- If a callback sequence is empty, TestStand does not invoke the Engine callback.
- Process models use the `Execution.EnableCallback` method to disable the `ProcessModelPostResultListEntry` and `SequenceFilePostResultListEntry` callbacks when the model does not need to process results on-the-fly for report generation or database logging.
- TestStand calls other Engine callbacks only when executing the `SequenceFileLoad` and `SequenceFileUnload` Engine callbacks. TestStand does not call Engine callbacks when executing the other Engine callbacks.

Front-End Callbacks

Front-End callbacks are sequences in the `FrontEndCallbacks.seq` file. Multiple user interface applications can call to share the same implementation for a specific operation. The `FrontEndCallback.seq` file TestStand installs in the `<TestStand>\Components\Callbacks\FrontEnd` directory contains one LoginLogout Front-End callback sequence. The TestStand Sequence Editor and default user interfaces call the LoginLogout callback.

Use Front-End callback sequences to implement operations so you can modify a Front-End callback without modifying the source code for the user interfaces or rebuilding the executables for the user interfaces. For example, to change how various user interfaces perform the login procedure, modify only the LoginLogout sequence in `FrontEndCallbacks.seq`.

To modify the default implementation of the Front-End callback or to create new Front-End callbacks, copy the `FrontEndCallbacks.seq` file from the `<TestStand>\Components\Callbacks\FrontEnd` directory to the `<TestStand Public>\Components\Callbacks\FrontEnd` directory and make any changes to copy of the file. When you copy installed files to modify, 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. You can use functions in the TestStand API to invoke the modified Front-End callback sequence file from each user interface application you create. However, because you cannot edit the source for the sequence editor, you cannot make the sequence editor call new Front-End callbacks you create.

Type Concepts

TestStand stores step type and data type definitions in files and in memory. You can modify TestStand types and use type version numbers to determine which version of the type to load. Use the Types window to create, modify, and examine step types and data types. Refer to Chapter 4, *Built-In Step Types*, for more information about the step types TestStand installs. Refer to Chapter 12, *Standard and Custom Data Types*, for more information about data types.

Storing Types in Files and Memory

TestStand files store the definition for each step type and data type the file uses. You can also specify that a file always saves the definition for a type, even if the file does not currently use the type. Because many files can use the same type, many files can contain the definition for the type. All sequence files, for example, might contain the definitions for the Pass/Fail Test step type and the Error standard data type.

TestStand allows only one definition for each uniquely named type in memory. The type can appear in multiple files, but only one underlying definition of the type exists in memory. If you modify the type in one file, the type definition updates in all loaded files. Refer to the *Types Window* section of this chapter for more information about viewing the types in memory and the files that reference the type.

Modifying Types

You can modify the built-in and custom properties of step types you create and custom data types you create. However, you cannot modify the built-in step types and standard data types TestStand installs.

Use the Copy and Paste context menu items to copy and rename an existing or built-in step type or standard data type in the Types pane of the Types window of the sequence editor.

When you modify a type, TestStand enables the Modified built-in property for the type. TestStand cannot automatically resolve type conflicts unless you disable the Modified property. To disable the Modified property, you typically increment the version number of the type on the Version tab of the Step Type Properties dialog box or on the Version tab of the Type Properties dialog box when you complete all the modifications to the type.

By default, the Before Saving Modified Types option on the Preferences tab of the Station Options dialog box is set to Prompt to Increment Version Types. This causes TestStand to launch the Modified Types Warning dialog box when you select **File»Save** and the sequence file or type palette contains types that are marked as modified. The prompt allows you to increment the type version or remove the modified mark on the type before saving, or save the type as modified. Refer to the *NI TestStand Help* for more information about the Before Saving Modified Types option on the Preferences tab of the Station Options dialog box and about the Modified Types Warning dialog box.

Type Versioning

When you complete edits to a type, increment the version number of the type. TestStand uses the version number to determine whether to load a type from a file when the type is already in memory and to determine which version of a type to use when the version numbers are different.

You can also specify the earliest TestStand version that can use a type to prevent the TestStand Engine from using the type if the version of the engine is earlier than the TestStand version you specify. If you enable this option and an earlier version of the engine attempts to load the type, TestStand ignores the type and loads the file only if an earlier version of the type already exists in memory.

Resolving Type Conflicts

If you load a file that contains a type definition and another type definition with the same name already exists in memory, TestStand compares the two type definitions, including all the built-in and custom subproperties in the types. If the types are identical, TestStand continues to use the type in memory.

If the types are not identical, TestStand attempts to resolve the type conflict. TestStand automatically selects the type with the greater version number when all of the following conditions exist:

- The **Modified** property is disabled for both types
- The **Always prompt the user to resolve the conflict** option on the Version tab of the Type Properties dialog box or the Step Type Properties dialog box is disabled for both types
- The **Allow Automatic Type Conflict Resolution** option on the Preferences tab of the Station Options dialog box does not restrict automatic type conflict resolution in this situation

If TestStand cannot automatically determine which type to use or if an execution is running and the type TestStand wants to use is located in the file being loaded, TestStand launches the Type Conflict in File dialog box, in which you can resolve the conflict. However, if TestStand is loading the file for execution and the type TestStand wants to use is located in the file being loaded, TestStand generates an error.

Type conflicts can occur when you use an earlier version of TestStand to open files saved in a newer version of TestStand. Typical differences include the addition of step type subproperties and altered flags. Select to use the earlier TestStand version of the type instead of the types from the file you are trying to open.

To prevent the altered version of a type from being used in or accidentally propagated to earlier TestStand version sequence files, enable the **Set Earliest TestStand Version that can Use this Type** option on the Version tab of the Step Type Properties dialog box and set the earliest version to the current version of TestStand.

In addition, when TestStand saves a sequence file as an earlier version of a TestStand sequence file, the TestStand Engine saves the types from the type palette files in the <TestStand>\Components\Compatibility\<VersionNumber> and <TestStand Public>\Components\Compatibility\<VersionNumber> directories with the sequence file. Place type palette files from earlier versions of TestStand in the <TestStand Public>\Components\Compatibility\<VersionNumber> directory to ensure that TestStand saves the correct version of the types with the sequence file.

Refer to the *NI TestStand Help* for more information about the Station Options dialog box, the Step Type Properties dialog box, the Type Conflict in File dialog box, and the Type Properties dialog box.

Types Window

Use the Types window in the sequence editor to view and edit step types, standard data types, and custom data types.

The View Types For pane of the Types window contains sections for type palettes, sequence files, and other items. When you select a file in the View Types For pane, the Types pane of the Types window lists the step types, standard data types, and custom data types used by or attached to the file. You can also display types for all loaded files. Use the Standard Data Types section to examine subproperties of the standard data types. Use the Custom Data Types section to create and modify custom data types. When you select **File»Save** in the Types window, TestStand saves the file you select in the Types pane.

Refer to the *NI TestStand Help* for more information about the Types window.

Type Palette Files

Type palette files contain step types, standard data types, and custom data types you want available in the sequence editor at all times. Drag a type to a type palette file in the Types window to ensure that the type is always available, even if the user manager, station globals, or any open sequence files do not use the type. Type palette files are located in the `<TestStand>\Components\TypePalettes` directory. Typically, you create new types in the `MyTypes.ini` type palette file in the `<TestStand Public>\Components\TypePalettes` directory or in a new type palette file you create.

You can distribute step types and data types you create to other computers by installing a type palette file in the `<TestStand Public>\Components\TypePalettes` directory. Use the `Install_` prefix in the name of the type palette files you want to install. At startup, TestStand searches the `TypePalettes` directory for type palette files with the `Install_` prefix. When TestStand finds a type palette file to install with a base filename that is not the same as any existing type palette file, TestStand removes the `Install_` prefix and adds the type palette to the type palette list. When TestStand finds a type palette file to install with a base filename that matches an existing type palette, TestStand merges the types from the install file into the existing type palette file and deletes the install file. This method is better than modifying the existing type palette file because this method is more modular and flexible for deployment and updates.

Sequence Files

Sequence files contain step types, standard data types, and custom data types the variables and steps in the file use. When you save the contents of the Sequence File window, TestStand saves the definitions of the types the sequence file uses in the sequence file itself.

When you create a new type in the Types pane for a sequence file, the type appears in the Insert Local, Insert Global, Insert Parameter, Insert Field, or Insert Step submenus only in that Sequence File window. To use a new type in other sequence files, copy or drag the new type to a type palette file because each type in a type palette file appears in the appropriate Insert submenu for all windows. You can also manually copy or drag the new type from one sequence file to another. Refer to the *NI TestStand Help* for more information about the Sequence File window.

Station Globals

Station globals contain standard and custom data types the station global variables use. When you save the contents of the Station Globals window, TestStand saves the definitions of the types station global variables use in the `StationGlobals.ini` file in the `<TestStand Application Data>\Cfg` directory. Refer to the *NI TestStand Help* for more information about the Station Globals window.

Because station globals are prone to race conditions and data validity issues, use queues and notifications for intrathread communication. Refer to Appendix B, *Synchronization Step Types*, and the *NI TestStand Help* for more information about Queue and Notification objects.

User Manager

All users and user profiles use the User standard data type. To add new privileges for all users and groups, add the privileges to the `NI_UserCustomPrivileges` type. When you save the contents of the User Manager window, TestStand saves the definitions of the types used to define users in the `Users.ini` file in the `<TestStand Application Data>\Cfg` directory. Refer to the *NI TestStand Help* for more information about the User Manager window. Refer to Chapter 7, *User Management*, for more information about using the TestStand User Manager.

Standard and Custom Data Types

When you insert variables, parameters, or step properties, you can select a data type to modify for the item. You can also create and modify custom data types to meet the needs of an application. Refer to Chapter 11, *Type Concepts*, for more information about types.

Using Data Types

The context menu of each window or pane in which you can insert a variable, parameter, or property includes an Insert item, as listed in Table 12-1.

Table 12-1. Creating Data Type Instances from Context Menus

Context Menu Item	Location of Context Menu	Item Inserted
Insert File Global	File Globals section of the Variables pane in the Sequence File window	Sequence file global variable
Insert Parameter	Parameters section of the Variables pane in the Sequence File window	Sequence parameter
Insert Local	Locals section of the Variables pane in the Sequence File window	Sequence local variable
Insert Station Global	Station Globals window	Station global variable
Insert User Insert Group	User Manager window	New object with the User data type
Insert Field	Types window	New element in an existing data type

With the exception of the Insert User and Insert Group items, all the context menu items in Table 12-1 provide a submenu from which you can select the following categories of data types:

- Simple data types TestStand defines, including the number, string, Boolean, and object reference data types.
- Container data type, in which you can add other data types. You can use an empty container as a parameter when you want to pass an object of any type to the sequence, in which case you also must turn off type checking for the parameter.
- Named data types, including all the custom named data types in type palette files or in the files you are currently editing. The submenu also includes standard TestStand named data types, such as Error, Path, Expression, and CommonResults. Refer to the [Using Standard Named Data Types](#) section of this chapter for more information about the standard named data types.
- An array of elements that all have the same data type.

If the submenu does not contain the data type you require, you must create the data type in the Types window and then select the new data type from the Type submenu of the Insert context menu. If the data type already exists in another window or pane, copy or drag the data type to a type palette file or to the file you are editing.

To create a parameter with a complex data type, first create the data type in the Types window and then select the complex data type from the Insert Parameter»Type submenu.

Specifying Array Sizes

Select an item from the **Array of** submenu of the Insert context menu to launch the Array Bounds dialog box, in which you can set and modify the array bounds. Figure 12-1 shows the settings for a three-dimensional array.

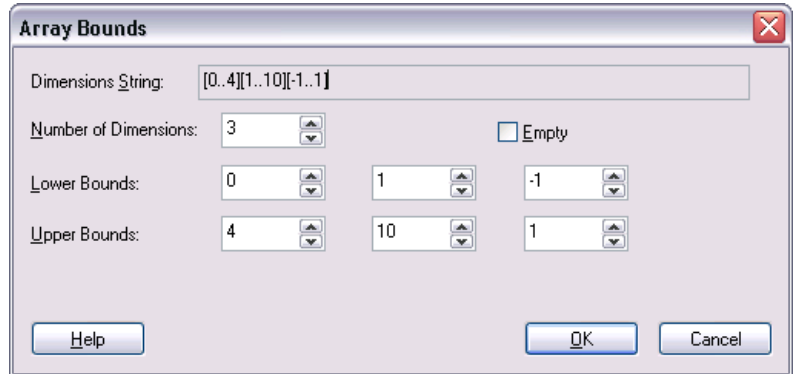


Figure 12-1. Array Bounds Dialog Box

The first and outermost dimension has five elements, with 0 as the minimum index and 4 as the maximum index. The second dimension has 10 elements, with 1 as the minimum index and 10 as the maximum index. The third and innermost dimension has three elements, with -1 as the minimum index and 1 as the maximum index.

After you create the variables, parameter, or property as an array, you can modify the array bounds by clicking the **Resize Array** button in the Name column of the list view to launch the Array Bounds dialog box. On the Types pane of the Types window, right-click the variable, parameter, or property, select **Properties** from the context menu, and click the **Bounds** tab of the Type Properties dialog box to modify the array bounds.

Dynamic Array Sizing

You can also resize an array during execution.

In an expression, use the `GetNumElements` and `SetNumElements` expression functions to obtain and modify the upper and lower bounds for a one-dimensional array. For multi-dimensional arrays or to change the number of dimensions in the array, use the `GetArrayBounds` and `SetArrayBounds` expression functions. The Operators/Functions tab of the Expression Browser dialog box includes documentation for each expression function. Refer to the *NI TestStand Help* for more information about the Expression Browser dialog box.

In a code module, use the `PropertyObject.GetDimensions` method and the `PropertyObject.SetDimensions` method to obtain or set the upper and lower bounds of an array or to change the number of dimensions in the array. Refer to the *NI TestStand Help* for more information about the

`PropertyObject.GetDimensions` method and the `PropertyObject.SetDimensions` method.

Empty Arrays

Enable the **Empty** option in the Array Bounds dialog box or on the Bounds tab of the Type Properties dialog box if you want the array to be empty when you start the execution. When you enable this option, the Upper Bounds control for each dimension dims. Defining an initially empty array is useful when you do not know the maximum array size the sequence requires during execution and when you want to save memory during the periods of execution when the sequence does not use the array.

Modifying Data Types and Values

With the exception of resizing arrays, you cannot change the internal structure of a variable, parameter, or property after you create it from a data type. You cannot change the data type setting or deviate from the data type.

You can, however, change the content of the data type itself. Changing the content of a data type affects all variables, parameters, and properties that use the data type.

Use the Value column on the Variables pane to modify the value of a variable, parameter, or property. For variables and properties, the data type value is the initial value TestStand uses when you start execution or call the sequence. For parameters, the data type value is the default value TestStand uses when you do not pass an argument value explicitly. On the Types pane of the Types window, right-click the variable, parameter, or property and select **Properties** from the context menu to launch the Type Properties dialog box, in which you can specify additional properties of the type. In general, if you make changes to property values in a type, the changes do not affect all instances of the type. Enable the **Apply Changes in this Dialog to all Loaded Instances of the Type** option in the Type Properties dialog box to apply the change to all loaded instances of the type. Refer to the *NI TestStand Help* for more information about using the Type Properties dialog box.

You can also rearrange variables, parameters, and properties in the Variables pane by dragging or copying the items you want to move. The order of variables and properties does not matter, but the order of parameters affects how you configure a Sequence Call step that invokes the sequence.

Object References

Object reference properties can contain references to .NET or ActiveX/COM objects. TestStand stores ActiveX references as an IDispatch pointer or an IUnknown pointer. If the variable, parameter, or property is an object reference, you can use the Release Object button, which displays only if the value of the variable, parameter, or property is non-zero, in the Value column on the Variables pane to release the reference.

You can set the reference value only by using an expression, by using a code module that uses the TestStand API, or by calling the TestStand API directly using the ActiveX/COM Adapter.

The value you assign to the object reference must be a valid object pointer. When you assign a non-zero value to an object reference, TestStand maintains a reference to the object for as long as the variable, parameter, or property contains the value. TestStand automatically releases the reference to the object when the variable, parameter, or property loses its scope. For example, if a sequence local variable contains a reference to an object, TestStand releases the reference when the call to the sequence completes. You can also release the reference to the object by assigning the variable, parameter, or property a new value or the constant `Nothing`. Do not release an object variable by assigning it a value of 0 because TestStand assigns a reference to the Numeric property for the reference object. Instead, use the constant `Nothing` to clear the reference. When you release all references to a .NET object, TestStand marks the object for garbage collection. When you release all references to an ActiveX/COM object, TestStand destroys the object.

If you have two reference properties, TestStand performs an equality comparison on the IUnknown pointers for ActiveX objects and the pointer values for .NET objects.

Using Standard Named Data Types

TestStand defines a set of standard named data types, such as `Error`, `CommonResults`, `Path`, and `Expression`. The only standard named data types you can modify are the `CommonResults` and the `NI_UserCustomPrivileges` types. With the `CommonResults` standard data type, you can add subproperties to the standard data types, but you cannot delete any of the built-in subproperties.

Error and CommonResults

TestStand inserts a Results property in every step you create. The Results property includes at least three subproperties—Error, Status, and CommonResults.

Steps use the Error subproperty to indicate run-time errors. The Error subproperty uses the Error standard data type, which is a container that includes three subproperties—Code, Msg, and Occurred. When a run-time error occurs in a step, the step sets the Code subproperty to a value that indicates the source of the error, the Msg subproperty to a string that describes the error, and the Occurred subproperty to `True`.

The CommonResults subproperty uses the CommonResults standard data type, which is an initially empty object. By adding subproperties to the CommonResults data type, you can add extra result information to all steps in a standard way. Newer versions of TestStand do not overwrite the subproperties you add to the CommonResults data type.

If you modify CommonResults without incrementing the type version number, you might see a type conflict when you open other sequence files, such as `FrontEndCallbacks.seq` when TestStand loads the LoginLogout Front-End callback before you log in or out. TestStand prompts you to increment the version number when you save changes to any data type or step type. National Instruments recommends modifying the CommonResults data type only if you want to make an architectural change to all step types that you use. Share the modified CommonResults data type and the step types that use the CommonResults data type only with systems on which you are certain no conflicting changes to CommonResults will be deployed. Refer to the [Type Versioning](#) section of Chapter 11, [Type Concepts](#), for more information about incrementing type version numbers.

Path

Use the Path standard data type to store a pathname as a string so TestStand can locate path values saved in variables and step properties when processing sequence files for deployment. National Instruments recommends always using relative paths when you prepare for deployment so TestStand can find files even if you install the files in a location on the target computer that is different than the location of the files on the development computer.

Expression

Use the Expression standard data type to store an expression as a string so TestStand can locate expression values saved in variables and step properties when editing sequence files.

Creating Custom Data Types

Complete the following steps to create a custom data type.

1. On the Types pane of the Types window, expand the **Custom Data Types** section.
2. Right-click and select **Insert Custom Data Type** from the context menu. You can also use the Copy and Paste context menu items to copy and rename an existing data type.
3. Select the data type you want from the submenu. Refer to the [Using Data Types](#) section of this chapter for more information about selecting a data type. If you select the Container type from the submenu, TestStand creates the data type without any fields in which you can insert additional data types.
4. Right-click the new data type and select **Properties** from the context menu to launch the Type Properties dialog box, in which you can specify the version number of the type and how to pass the data type to LabVIEW, LabWindows/CVI, and .NET code modules.

When you create new data types, use unique names to avoid conflicts with the default names TestStand uses. For example, begin new custom data type names with a unique ID, such as a company prefix.

Properties Common to All Data Types

TestStand defines many built-in data type properties common to all data types. You can examine and modify the values of the built-in data type properties in the Types window. Right-click a data type and select **Properties** from the context menu to launch the Type Properties dialog box, which contains the following tabs:

- **General tab**—Use this tab to change the value, numeric format, flags, and comments for the property. Click the **Advanced** button to launch the Edit Flags dialog box, in which you can modify the property flags. Typically, you need to configure property flags only when you develop a relatively sophisticated custom data type. Refer to the *NI TestStand Help* for more information about the Edit Flags dialog box. Refer to the

PropertyFlags Constants and the *PropertyObjTypeFlags Constants* topics in the *NI TestStand Help* for a description of each property flag constant in the TestStand API.

- **Bounds tab**—Use this tab to specify array sizes. This tab is visible only for array data types.
- **Version tab**—Use this tab to edit the version information for the data type, to determine if the data type is modified, to specify how TestStand resolves data type conflicts, and to specify the earliest version of TestStand that can use the type when you save the file for an earlier version of TestStand.
- **Cluster Passing tab**—Use this tab to specify how TestStand passes instances of the data type as a cluster to LabVIEW code modules.
- **C Struct Passing tab**—Use this tab to specify how TestStand passes instances of the data type as a structure to functions and methods in C/C++ DLL code modules.
- **.NET Struct Passing tab**—Use this tab to define how TestStand passes instances of the data type as a structure to methods and properties in .NET assemblies.

Refer to the *NI TestStand Help* for more information about each tab in the Type Properties dialog box.

Custom Properties of Data Types

You can add any number of fields to a container data type or container data type subproperty you create. On the Types pane of the Types window, expand the data type or data type subproperty, right-click, select **Insert Field** from the context menu, and select a data type from the submenu to add fields to a container property in a new or existing data type. Right-click the field and use the context menu to cut, copy, paste, delete, and rename fields.

Custom Step Types

You can create custom step types to meet the needs of an application. Refer to Chapter 11, *Type Concepts*, for more information about types.

Custom step types differ from the step templates you store in the Templates list on the Insertion Palette. Custom step types define standard functionality for a class of steps. Step templates are preconfigured instances of step types you typically use, such as calls to frequently used code modules. Changes you make to step types can affect step instances previously inserted into sequences, but changes you make to step templates do not affect steps previously inserted into sequences. Refer to the *NI TestStand Help* for more information about the Templates list on the Insertion Palette.

Creating Custom Step Types

Complete the following steps to create a custom step type.

1. On the Types pane of the Types window, expand the **Step Types** section.
2. Right-click and select **Insert Step Type** from the context menu. You can also use the Copy and Paste context menu items to copy and rename an existing step.

When you create new step types, use unique names to avoid conflicts with the default names TestStand uses. For example, begin new custom step type names with a unique ID, such as a company prefix.

3. Right-click the new step and select **Properties** from the context menu to launch the Step Type Properties dialog box.
4. Click the **Menu** tab and specify the menu item name for the new step.
5. Click the **General** tab and specify the default name for new steps you create from the new type and specify the description expression for those steps.
6. Click the **Substeps** tab, select an adapter, click **Add**, and select the type of step to create substeps. Use the Substep Info section of the Substeps tab to specify the menu item name of Edit steps.

Properties Common to All Step Types

TestStand defines many built-in step type properties common to all step types.

The class step type properties exist only in the step type itself. TestStand uses the class step type properties to define how the step type works for all step instances. Step instances do not contain copies of the class step type properties.

The instance step type properties exist in each step instance. Each step you create with the step type includes a copy of the instance step type properties. TestStand uses the value you specify for an instance step type property as the initial value of the property in each new step you create.

After you create a step, you can change the values of the properties for a step type instance, but these changes do not propagate to other step type instances. When you create a custom step type, you can prevent users from changing the values of specific instance step type properties in the steps they create. For example, you can use the Edit substep of a step type to set the Status Expression for the step, in which case you do not want users to explicitly change the Status Expression value. Some of the built-in step types, such as the Numeric Limit Test and the String Value Test, prevent you from changing the value of the instance step type properties.

Step Type Properties Dialog Box

You can examine and modify the values of the built-in step type properties in the Types window. Right-click a step and select **Properties** from the context menu to launch the Step Type Properties dialog box. TestStand uses the values on the Default Run Options, Default Post Actions, Default Expressions, Default Loop Options, Default Switching, and Default Synchronization tabs as the initial values for new steps you create. These tabs have the same appearance and behavior as the Run Options, Post Actions, Expressions, Looping, and Synchronization panels on the Properties tab on the Step Setting pane for a step instance in the sequence editor. Changes to the default values do not automatically propagate to existing steps of this type. You can enable the Apply Changes in this Dialog to all Loaded Steps of this Type option to propagate changes to steps of this type currently in memory, but unloaded files are not updated. Refer to the *NI TestStand Help* for more information about these tabs. The General, Menu, Substeps, Disable Properties, Code Templates, and Version tabs include class step type properties.

General Tab

Use the General tab to specify a name, description, and comment for the step type. You can also specify the default module adapter and the default code module the step type calls. However, after you create an instance step type, you can use the Properties tab of the Step Settings pane in the sequence editor or the Step Properties dialog box in a user interface to change the adapter and code module call. If you want code module changes to propagate to all instances of the step type, you must change all instances of the step type to use the <None> adapter so the step does not call a code module, and create a Post-Step substep for the step type and call the code module from this substep instead of specifying a default adapter and code module. You must also enable the Specify Module option on the Disable Properties tab if you do not want sequence developers to change or edit the default code module call. Refer to the [Substeps Tab](#) section of this chapter for more information about substeps. Refer to the [Disable Properties Tab](#) section of this chapter and the *NI TestStand Help* for more information about the Disable Properties tab.

Click the **Advanced** button and select **Flags** to launch the Edit Flags dialog box, in which you can modify the property flags. Typically, you need to configure property flags only when you develop a relatively sophisticated custom step type. Refer to the *NI TestStand Help* for more information about the Edit Flags dialog box. Refer to the *PropertyFlags Constants* and the *PropertyObjTypeFlags Constants* topics in the *NI TestStand Help* for a description of each property flag constant in the TestStand API.

Click the **Advanced** button and select **Block Structure** to launch the Block Structure dialog box, in which you can specify if instances of this step type affect the block structure in a sequence. The TestStand Flow Control step types, such as If, ElseIf, and End, use these built-in properties. Refer to the *NI TestStand Help* for more information about the Block Structure dialog box.

Menu Tab

Use the Menu tab to specify the menu item name that appears for the step type in the Insert Step context menu. Use the Step Type Menu Editor to organize the Step Types list of the Insertion Palette and the Insert Step submenu of the Steps pane context menu. Refer to the *NI TestStand Help* for more information about the Step Type Menu Editor. Refer to the [Using Step Types](#) section of Chapter 4, [Built-In Step Types](#), for more information about the Insertion Palette.

Substeps Tab

Use the Substeps tab to specify Pre-Step, Post-Step, Edit, and Custom substeps for the step type. Substeps use substep code modules to define standard actions, other than calling the step code module, TestStand performs for all instances of the step type.

After you add a substep to a step type, use the Specify Module button to configure the substep module call. For each step that uses the step type, TestStand calls the same substep modules with the same arguments. You cannot add or remove substeps or otherwise alter the substep module call the substep performs when you configure a step instance.

Although you can specify any number of substeps for a step type, the list of substeps is not a sequence, and substeps do not have preconditions, post actions, or other execution options. The order in which Pre- and Post-Step substeps execute is the only execution option you specify.

TestStand calls the Pre-Step substep before calling the step code module. For example, a Pre-Step substep might call a substep code module that retrieves measurement configuration parameters and stores those parameters in step properties the step code module uses.

TestStand calls the Post-Step substep after calling the step code module. For example, a Post-Step substep might call a substep code module that compares the values the step code module stored in step properties against limit values the Edit substep stored in other step properties. You can have multiple Post-Step substeps that execute in order.

TestStand calls an Edit substep when you select the substep menu item from the Steps pane context menu. On the Substeps tab, select the Edit substep and click the **Rename** button to specify the name of the substep menu item and the caption of the button on the Step Type Edit tab of the Step Settings pane.

The Edit substep typically calls a substep code module that launches a dialog box in which you can edit the values of the custom step properties. For example, an Edit substep might launch a dialog box in which you specify the high and low limits for a test. The Edit substep might then store the high and low limit values as step properties.

Dialog boxes the Edit substep launches must be modal. Refer to the `<TestStand Public>\Examples\ModalDialogs` directory for LabVIEW and MFC examples of modal dialog boxes.



Note You can initialize threads within TestStand executions to use the single-threaded apartment model or the multi-threaded apartment model. TestStand executes Edit substeps only in threads initialized using the single-threaded apartment model so the substep can open windows that contain ActiveX controls.

Typically, TestStand does not call custom substeps during an execution. Use the TestStand API to invoke a custom substep from a code module or user interface. You can create a custom substep named `OnNewStep` for TestStand to call each time you create a new step of that type. For example, the built-in If step type uses an `OnNewStep` substep to insert an End step.

The `<TestStand>\Components\StepTypes` directory includes source code for many of the substep modules the built-in step types use. To modify the installed step types or to create a new step type, copy the step type source code from the `<TestStand>\Components\StepTypes` directory to the `<TestStand Public>\Components\StepTypes` directory and make changes to the copy of the source code. When you copy installed files to modify, 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.

Disable Properties Tab

Use the Disable Properties tab to prevent sequence developers from modifying the settings of built-in instance step type properties in individual steps. Each option on the Disable Properties tab represents one built-in instance property or a group of built-in instance properties. When you enable an option, you prevent sequence developers from modifying the value of the corresponding property or group of properties for all step instances.



Note When you create new steps, TestStand uses the default values of built-in step type properties as the initial values for the new steps. Subsequent changes to these default property values do not automatically propagate to existing step type instances, even when you enable the corresponding option on the Disable Properties tab.

Code Templates Tab

Use the Code Templates tab to associate one or more code templates with the step type. A code template is a set of source files that contains skeleton code to serve as a starting point for developing code modules for steps that use the step type. TestStand uses the code template when you click the **Create Code** button on the Module tab of the Step Settings pane for a step in the sequence editor.

You can use the default TestStand code templates for any step type, and you can customize code templates for individual step types. For example, for the Numeric Limit Test step type, you might want to include code for accessing the high- and low-limit properties in a step.

Template Files for Different Development Environments

Because different module adapters require different types of code modules, code templates typically correspond to a particular programming language in a specific development environment. The `<TestStand>\CodeTemplates` directory includes the default code templates for each development environment, as shown in Table 13-1.

Table 13-1. Default Code Templates in `<TestStand>\CodeTemplates`

Subdirectory Name	Template Description
Default_Template	Legacy default template
DefaultC++.NET	Default template for C++ in Microsoft Visual Studio .NET 2003 or Visual Studio 2005
DefaultCSharp.NET	Default template for C# in Visual Studio
DefaultCVI	Default template for C in LabWindows/CVI
DefaultHTB72_Template	Default template for HTBasic 7.2
DefaultHTB80_Template	Default template for HTBasic 8.0
DefaultLabVIEW	Default template for LabVIEW
DefaultVB.NET	Default template for Microsoft Visual Basic .NET
DefaultVC++_Template	Default template for C++ in Visual Studio

Each subdirectory includes the source file for the module adapter and a .ini file that contains parameter information and a description string TestStand displays for the code template. TestStand uses the directory names from the <TestStand>\CodeTemplates or <TestStand Public>\CodeTemplates directory as the code template name to display in the Code Templates tab of the Step Type Properties dialog box.

Code templates for the LabVIEW, LabWindows/CVI, and C/C++ DLL Adapters can have any number of parameters compatible with the data types you can specify on the Module tab for those adapters.

When TestStand uses a code template for a DLL to create skeleton code, it compares the parameter list in the source file to the parameter information on the Module tab. If these two sources of information do not match, TestStand prompts you to select which prototype to use for the skeleton code. If you use the prototype from the template source file, TestStand updates the Module tab to match the prototype in the template source file. However, the template source file does not contain sufficient information for TestStand to update the Value controls for the parameters on the Module tab. Use the Parameter Name/Value Mappings section in the Edit Code Template dialog box to specify entries for TestStand to place in the Value controls. TestStand stores the parameter values in the .ini file in the template subdirectory.

Legacy Code Templates

In TestStand 3.5 and earlier, code template directories contain source files for multiple development environments. For example, a legacy code template directory might include one .c file for the LabWindows/CVI Adapter and multiple VIs for the LabVIEW Adapter, where each VI corresponds to the different combinations of parameter options users can set in the Edit LabVIEW VI Call dialog box. TestStand includes these legacy code templates to provide backward compatibility with previous versions of TestStand. The [TemplateType] section of the config.ini file in each code template directory includes Type = "Legacy" for legacy code templates.

Legacy code templates for the LabVIEW Adapter always specify **Test Data** and **Error Out** clusters as parameters. The VIs for each LabVIEW Adapter legacy code template specify various combinations of the **Input Buffer**, **Invocation Info**, and **Sequence Context** parameters. When TestStand uses a legacy LabVIEW template VI to create skeleton code, it selects the correct VI to use according to the current settings in the Optional Parameters dialog box in TestStand 3.5 and earlier.

Legacy code templates for the LabWindows/CVI Adapter always specify two parameters—a pointer to a **tTestData** structure and a pointer to a **tTestError** structure. When TestStand uses a legacy LabWindows/CVI template module to create skeleton code, it validates the function prototype in the template module against this requirement. TestStand reports an error if the prototype is incorrect.

Creating and Customizing Code Template Files

Click the **Create** button on the Code Templates tab to launch the Create Code Templates dialog box. TestStand copies the files for the existing code template you select into a new subdirectory in the <TestStand Public>\CodeTemplates directory based on the code template name you specified in the Create Code Templates dialog box. You can then customize the code template files in the new <TestStand Public>\CodeTemplates directory.

For example, you can include example code that shows users how to access the custom properties of the step. For most environments, you can add a value parameter to pass the information from TestStand. You can also show how to obtain the high- and low-limit properties in a LabVIEW or LabWindows/CVI code template for a Numeric Limit Test step by customizing the prototype for the code module to specify the high and low limits as value parameters. As another example, you might want to show how to return a measurement value from a code module. For the LabVIEW, LabWindows/CVI, and C/C++ DLL Adapters, you can customize the prototype in the code template by specifying the measurement as a reference parameter.

Multiple Code Templates per Step Type

You can specify more than one code template for a step type. For example, you might want to have code templates that contain example code for conducting the same type of tests with different types of instruments or data acquisition boards. When a step type has multiple code templates and you click the **Create Code** button on the Module tab, TestStand prompts you to select from a list of templates or uses the template you selected on the Module tab if it exists.

Version Tab

Use the Version tab to edit the version information for the data type, to determine if the data type is modified, to specify how TestStand resolves data type conflicts, and to specify the earliest version of TestStand that can use the type when you save the file for an earlier version of TestStand.

Custom Properties of Step Types

You can add any number of custom properties in a step type you create. Each step you create using the step type includes the custom properties you create. On the Types pane of the Types window, expand the step type, right-click, select **Insert Field** from the context menu, and select a data type to add fields to a step type. Right-click the field and use the context menu to cut, copy, paste, delete, and rename fields.

Backward Compatibility

When you modify custom step types, avoid making changes that might jeopardize backward compatibility. Ensure that previously configured steps behave properly when you execute the steps using the modified custom step type. Also ensure that new step instances based on the modified step type behave properly when you save a sequence file to an earlier version of TestStand that uses the original custom step type.

Do not rename custom step type properties or change the functionality of existing properties. For example, if you have an existing property that performs a specified task and you later decide you want the property to do something completely different in a future instance of the step type, you break backward compatibility. When you create new properties, provide default values that preserve the functionality of previously created steps. When you extend enumerated values, do not change the functionality of previously used values.

Deploying TestStand Systems

The TestStand Deployment Utility helps to simplify the complex process of deploying a TestStand system by automating many of the steps involved, including collecting sequence files, code modules, and support files for the test system and creating an installer for the files. The versions of the TestStand Deployment Utility and the TestStand development system must match.

TestStand System Components

The following components work together to create the entire TestStand system:

- TestStand Engine and supporting files
- LabVIEW and LabWindows/CVI Run-Time Engines
- Process models and supporting files
- Step types and supporting files
- Configuration files
- User interface applications
- Workspace files
- Sequence files
- Code modules and supporting files
- Hardware drivers

When you deploy a TestStand system from a development computer to a target computer, you must deploy all the components the system uses to the target computer, including files you call dynamically.

Setting Up the TestStand Deployment Utility

To deploy a TestStand test system using the TestStand Deployment Utility, you must identify the components to deploy, determine if you need to create an installer for the system, create a system workspace file, if necessary, and configure and build the deployment.

Identifying Components to Deploy

You can deploy TestStand components in the `<TestStand Public>` directory and you can deploy a TestStand workspace file and its dependent files, including sequence files, code modules, and so on. Additionally, you can use the TestStand Deployment Utility to create an installer that also includes the TestStand Engine, hardware drivers, and components in the `<TestStand>` subdirectories. Refer to the [TestStand Directory Structure](#) section of Chapter 8, [Customizing and Configuring TestStand](#), for more information about TestStand directories.

Make sure the files you want to deploy use unique filenames because using files with the same name can cause the deployment utility to locate incorrect files, which can result in incorrect behavior.

Determining If You Need to Create an Installer

If you plan to deploy the TestStand Engine and the TestStand components in the `<TestStand>` subdirectories, you must use the TestStand Deployment Utility to create an installer.

You do not need to use the deployment utility to create an installer if you plan to use a third-party installer development tool, such as Wise or InstallShield, or if you plan to use a source code or revision control system to deploy the system files to target computers.

Creating a System Workspace File

Before you deploy sequence files and code modules, you must create a workspace file that contains all the sequence files the test system might execute, files you do not store in a `<TestStand Public>` directory, files that sequence files do not reference directly, such as support files code module DLLs require, and files the sequence calls dynamically. The deployment utility analyzes the sequence files to determine which files the sequence files reference, such as code module files.

If you use the TestStand Deployment Utility to deploy only the TestStand Engine or the components in the `<TestStand Public>` subdirectories, you do not need to create a workspace file for the test system.

Refer to the [Workspaces](#) section of Chapter 2, [Sequence Files and Workspaces](#), for more information about TestStand workspace files.

Configuring and Building the Deployment

Select **Tools»Deploy TestStand System** in the sequence editor or in a user interface in Editor Mode to launch the TestStand Deployment Utility to configure the settings for deploying a test system, including the components to install and the installer settings. Refer to the *NI TestStand Help* for more information about the TestStand Deployment Utility.

Building a Deployment

The TestStand Deployment Utility collects and filters files to include in the deployment, processes VIs and sequence files, and packages National Instruments hardware drivers and components to build a deployable test system.

Collecting Files

When deploying a workspace file, the deployment utility analyzes the workspace for any dependent files. For example, the deployment utility searches the steps in every sequence of a sequence file in the workspace file to find the referenced code modules and continues recursively searching until the utility analyzes all the files in the workspace hierarchy.

The TestStand Deployment Utility does not automatically deploy .NET or ActiveX/COM code modules. You must manually add these code modules and supporting files to the workspace file or install the files separately on the target computer.

Because distributing every file sequences use might be problematic, the deployment utility includes a filtering function that removes potentially unwanted files. For example, if steps in a sequence call functions in Windows system DLLs, the deployment utility does not deploy those DLLs to the target computer.

Edit the `Filter.ini` file in the `<TestStand Application Data>\Cfg` directory to define the files the deployment utility automatically excludes from any deployment package it creates. By default, the deployment utility does not deploy any files in the `<TestStand>\Bin` or `<TestStand>` subdirectories or any `.exe` or `.dll` files in the `<Windows>` or `<Windows>\System32` directories.

You can add automatically excluded files to a workspace file, but do so with caution to prevent incompatibility issues. For example, deploying a Windows system DLL from a development computer running Windows XP to a target computer running Windows 2000 might result in DLL version incompatibility issues.

Processing VIs

You must have the LabVIEW Development System installed on the development computer for the TestStand Deployment Utility to process VIs.

The deployment utility analyzes the LabVIEW VIs it deploys to determine their complete hierarchies, including all subVIs, DLLs, external subroutines, run-time menus, Express VI configurations, and help files the VIs might reference. The deployment utility packages these VIs and their hierarchies to ensure that the VIs can run on computers that do not have the LabVIEW Development System installed. If the VIs you want to deploy call other VIs dynamically using VI Server, you must add the dynamically called VIs manually to the workspace file.

Refer to the *Building a TestStand Deployment with LabVIEW 8.0* section of Appendix A, *Using LabVIEW 8.x with TestStand*, of the *Using LabVIEW with TestStand* manual for more information about restrictions for deploying LabVIEW 8.0 or later VIs.

Processing Sequence Files

The TestStand Deployment Utility also processes sequence files to remove absolute paths because functional absolute paths on a development computer might be invalid on the target computer, especially if the two computers use different base installation directories. The deployment utility changes absolute path references in sequence files to relative path references that initiate from one of the following search directories:

- Current sequence file directory
- TestStand installation directory
- Windows\System32 directory
- Windows directory
- <TestStand Public> directory

If the files do not reside in one of these directories, the deployment utility does not change the absolute paths, which might not resolve correctly on the target computer.

Refer to the [Search Paths](#) section of Chapter 5, [Module Adapters](#), for more information about TestStand search directories.

Installing National Instruments Components

Use the TestStand Deployment Utility to package National Instruments hardware drivers and other components, such as run-time engines, in deployment installers. Click the **Drivers and Components** button on the **Installer Options** tab of the TestStand Deployment Utility to launch the Drivers and Components dialog box.

The Drivers and Components dialog box lists only components on the development computer you installed from the NI Device Driver CD that ships with TestStand or from a later version of the driver CD. The components you select contain only the product features you installed on the development computer.

Refer to the *NI TestStand Help* for more information about the Drivers and Components dialog box.

Guidelines for Successful Deployment

Use the following guidelines to ensure a successful deployment process:

- Always use unique filenames because using files with the same name can cause the deployment utility to locate incorrect files, which can result in incorrect behavior. The TestStand Deployment Utility returns an error when LabVIEW 8.0 or later VIs or subcomponents, such as DLLs, use the same filename. Before you create a deployment, you must ensure that all sequences you include in the deployment image reference unique VI and DLL files.
- Use relative paths and search paths so TestStand can find files even if you install the files in a location on the target computer that is different than the location of the files on the development computer. Refer to the [Configuring Search Paths for Deployment](#) section of Chapter 5, [Module Adapters](#), for more information about configuring TestStand search directories for deployment.

- Manually add dynamically referenced files to the workspace. Dynamically referenced files include any sequences an expression specifies, property loader files expressions specify, VIs you call using VI Server, and dynamically loaded DLLs.
- Manually add supporting DLLs code modules require to the workspace. Do not add any DLLs that are part of TestStand or the operating system.
- Redeploy the system if you edit any deployed system files because the deployed system might not function properly otherwise.
- Install the complete drivers from the NI Device Driver CD on development computers where you intend to use the TestStand Deployment Utility to ensure that any deployments you build on the development computer can access the most complete version of the driver software.

Refer to the *Building a TestStand Deployment with LabVIEW 8.0* section of Appendix A, *Using LabVIEW 8.x with TestStand*, of the *Using LabVIEW with TestStand* manual for more information about restrictions for deploying LabVIEW 8.0 or later VIs.

Common Deployment Scenarios

To complete the following examples that describe how to use the TestStand Deployment Utility in common deployment scenarios, you need one development computer that contains a complete installation of TestStand and one target computer. To run the TestStand Sequence Editor or User Interface application on the target computer, you must activate an appropriate license or run the application in evaluation mode. Refer to the *TestStand Licensing Options* topic in the *NI TestStand Help* for more information about the available TestStand license options.

Deploying the TestStand Engine

Complete the following steps to deploy the TestStand Engine.

1. Select **Tools»Deploy TestStand System** in the sequence editor to launch the TestStand Deployment Utility.
2. On the **System Source** tab, enable the **Deploy Files in TestStand User Directories** option to collect files from the <TestStand Public> directories. When you select this option, the deployment utility distributes to the target computer any component file customizations, such as process models, step types, and language strings, you saved in the <TestStand Public> subdirectories.

3. Use the **Location of Deployable Image** field to specify the directory to which the deployment utility copies an image of the system.
4. On the **Installer Options** tab, enable the **Install TestStand Engine** option and click the **Engine Options** button to launch the TestStand Engine Options dialog box, in which you select the TestStand components to include in the installer.
5. Expand the **TestStand Development Components** section and enable the **TestStand Sequence Editor** option to include the application in the engine installation. Click **OK** to accept the new settings and close the dialog box.

Refer to the [Distributing a User Interface](#) section of this chapter for more information about including a custom user interface in a deployment.

6. Click **Save** and save the build as `EngineInstaller.tsd`.
7. Click the **Build** button to create the installer.
8. To use the installer, copy all the files from the directory you specified on the System Source tab to a CD or to a shared directory on a network.
9. On the target computer, insert the CD or connect to the network and run the `setup.exe` application to start the installer.
10. When the installation completes, select **Start»All Programs»National Instruments»TestStand x.x»Sequence Editor** to verify that the TestStand Engine installed correctly. Activate a license if the sequence editor prompts you to do so.

Distributing Tests from a Workspace

Complete the following steps to distribute tests from a workspace.

1. Select **Tools»Deploy TestStand System** in the sequence editor to launch the TestStand Deployment Utility.
2. On the **System Source** tab, enable the **Deploy Files from TestStand Workspace File** option and use the **Location of Deployable Image** field to specify the directory to which the deployment utility copies an image of the system.
3. Click the **File Browse** button located next to the Workspace File Path control to browse to the `<TestStand Public>\Examples\Deployment` directory and select the `test.tsw` workspace file. Click **Open**.
4. Click the **Distributed Files** tab. A dialog box launches to request permission to analyze the source files. Click **Yes** for the deployment utility to analyze the workspace file and dependent files.

5. Click the **Build Status** tab and review the Status Log to check for analysis result warnings.
6. Disable the **unused.dll** option to remove it from the distribution because the example test system does not use this DLL.
7. On the **Installer Options** tab, enable the **Install TestStand Engine** option and click the **Engine Options** button to launch the TestStand Engine Options dialog box, in which you select the TestStand components to include in the installer.
8. Expand the **TestStand Development Components** section and enable the **TestStand Sequence Editor** option to include the application in the engine installation. Click **OK** to accept the new settings and close the dialog box.

Refer to the *Distributing a User Interface* section of this chapter for more information about including a custom user interface in a deployment.
9. Click **Save** to save the build as `test.tsd`.
10. Click the **Build** button to create the installer.
11. To use the installer, copy all the files from the directory you specified on the System Source tab to a CD or to a shared directory on a network.
12. On the target computer, insert the CD or connect to the network and run the `setup.exe` application to start the installer.
13. When the installation completes, select **Start»All Programs»National Instruments»TestStand x.x»Sequence Editor** to verify that the TestStand Engine installed correctly. Activate a license if the sequence editor prompts you to do so.
14. Verify the installation by loading and running `<TestStand Public>\Examples\Deployment\test.seq`.

Adding Dynamically Called Files to a Workspace

Complete the following steps to add dynamically called files to a workspace.

1. Select **Tools»Deploy TestStand System** in the sequence editor to launch the TestStand Deployment Utility.
2. On the **System Source** tab, enable the **Deploy Files from TestStand Workspace File** option and use the **Location of Deployable Image** field to specify the directory to which the deployment utility copies an image of the system.

3. Click the **File Browse** button located next to the Workspace File Path control to browse to the <TestStand Public>\Examples\Deployment directory and select the `Dynamically_called_sequence.tsw` workspace file. Click **Open**.
4. Click the **Distributed Files** tab. A dialog box launches to request permission to analyze the source files. Click **Yes** for the deployment utility to analyze the workspace file and dependent files.
5. Click the **Build Status** tab and review the Status Log, which reports a warning that an expression calls a sequence file you might need to add to the workspace.
6. Click the **Distributed Files** tab and notice that `Dynamic.seq` is missing in the Distributed Files list.
7. In the sequence editor, load the <TestStand Public>\Examples\Deployment\`Dynamically_called_sequence.tsw` workspace file.
8. Add <TestStand Public>\Examples\Deployment\`Dynamic.seq` to the workspace file and save the changes.
9. On the **Distributed Files** tab of the TestStand Deployment Utility, click the **Analyze Source Files** button to analyze the modified workspace file. Notice that the Distributed File list now includes `Dynamic.seq`.
10. Click the **Build Status** tab and review the Status Log, which reports a warning about an expression calling a sequence file. You can ignore the warning because you just added the correct sequence to the workspace.
11. On the **Installer Options** tab, enable the **Install TestStand Engine** option and click the **Engine Options** button to launch the TestStand Engine Options dialog box, in which you select the TestStand components to include in the installer.
12. Expand the **TestStand Development Components** section and enable the **TestStand Sequence Editor** option to include the application in the engine installation. Click **OK** to accept the new settings and close the dialog box.

Refer to the *Distributing a User Interface* section of this chapter for more information about including a custom user interface in a deployment.
13. Click **Save** to save the build as `Dynamic.tsd`.
14. Click the **Build** button to create the installer.

15. To use the installer, copy all the files from the directory you specified on the System Source tab to a CD or to a shared directory on a network.
16. On the target computer, insert the CD or connect to the network and run the `setup.exe` application to start the installer.
17. When the installation completes, select **Start»All Programs»National Instruments»TestStand x.x»Sequence Editor** to verify that the TestStand Engine installed correctly. Activate a license if the sequence editor prompts you to do so.
18. Verify the installation by loading and running `<TestStand Public>\Examples\Deployment\Call_sequence_dynamically.seq`.

Distributing a User Interface



Note You must install on the target computer the LabVIEW or LabWindows/CVI Run-Time Engine version that corresponds to the development environment version you use to create user interfaces. Refer to the `<TestStand>\Doc\Readme.html` file for information about the run-time engine versions TestStand installs.

Complete the following steps to distribute a user interface.

1. In the sequence editor, select **File»New Workspace File** to create and save a new workspace file as `Deploy User Interface.tsw`.
2. Right-click the Workspace window and select **Insert New Project into Workspace** from the context menu. Save the project as `User Interface.tpj`.
3. Right-click the **User Interface** project and select **Add Files to Project** from the context menu.
4. In the file browse dialog box, browse to the `<TestStand Public>\UserInterfaces\Simple\CVI` directory and change the **Files of Type** setting to **All Files (*.*)**.
5. Select `TestExec.exe` and `TestExec.uir` and click **Add**. If TestStand prompts you to resolve the path, select **Use a relative path for the file you selected** and enable the **Apply to All** option.
6. Click **OK** to close the dialog boxes.
7. Save the workspace file.
8. Select **Tools»Deploy TestStand System** in the sequence editor to launch the TestStand Deployment Utility.

9. On the **System Source** tab, enable the **Deploy Files From TestStand Workspace File** option and use the **Location of Deployable Image** field to specify the directory to which the deployment utility copies an image of the system.
10. Click the **File Browse** button located next to the Workspace File Path control to browse to the workspace file you saved in step 8. Click **Open**.
11. Click the **Distributed Files** tab. A dialog box launches to request permission to analyze the source files. Click **Yes** for the deployment utility to analyze the workspace file and dependent files.
12. Select `TestExec.exe` in the Distributed Files list. The File Properties section to the right of the Distributed Files list updates to reflect this selection.
13. In the File Properties section of the Distributed Files tab, enable the **Create Program Item** option and enter `Simple CVI UI` in the neighboring string field to add a shortcut menu item for `TestExec.exe`.
14. On the **Installer Options** tab, enable the **Install TestStand Engine** option.
15. Click **Save** to save the build as `SimpleCVIUI.tsd`.
16. Click the **Build** button to create the installer.
17. To use the installer, copy all the files from the directory you specified on the System Source tab to a CD or to a shared directory on a network.
18. On the target computer, insert the CD or connect to the network and run the `setup.exe` application to start the installer.
19. When the installation completes, select **Start»All Programs»<My TestStand System>>Simple CVI UI** to verify the installation.

Sequence File Translators

TestStand uses custom sequence file translators to load test description files saved in a custom format, such as text or XML. The translator reads the content of the custom sequence file, translates the content to a TestStand sequence file, and opens the TestStand sequence file in the sequence editor or a user interface. A custom sequence file translator can use predefined step types to simplify the mapping of common operations the custom file format defines to TestStand steps in sequence files.

Within the sequence editor or user interface, you can perform all typical operations TestStand sequence files support, such as executing and debugging sequences, diffing files, adding custom sequence files to workspaces, and deploying custom sequence files. However, you cannot automatically save changes you make to the sequence file in the sequence editor or user interface back to the custom sequence file format. You must make all changes to the custom sequence file directly.

You can create sequence file translators in various development environments, use versioning schemes with custom files, and deploy translators with TestStand. Refer to the `<TestStand Public>\Examples\SequenceFileTranslators` directory for example custom sequence files and translators.

Using a Sequence File Translator

TestStand can load custom sequence files if an existing translator can read and convert the file into a TestStand `SequenceFile` object. Translators are Windows DLLs that export callback functions TestStand uses to translate files. Refer to the [Creating a Translator DLL](#) section of this chapter for more information about creating translators. Refer to the *NI TestStand Help* for a complete list of callback functions the DLL must implement.

When an application loads the TestStand Engine, TestStand loads the DLLs that export the required callback functions from the `<TestStand>\Components\Translators` directory or the `<TestStand Public>\Components\Translators` directory. To create new translator DLLs, add the project for the translator to the `<TestStand Public>\`

Components\Translators directory. Ensure that the project saves the DLL to the <TestStand Public>\Components\Translators directory. Storing new and customized files in the <TestStand Public> directory ensures that newer installations of the same version of TestStand do not overwrite the customizations and ensures that uninstalling TestStand does not remove the files you customize.

A translator DLL can contain one or more translators. When TestStand loads a translator DLL, TestStand uses the callback functions of the DLL to obtain information about the translators the DLL contains. TestStand calls the CanTranslate callback function to determine if the DLL contains a translator that recognizes a file. The callback returns the index of the translator that recognizes the file after examining the extension of the file and the content of the file, typically the file header. Most of the callback functions the translator DLL implements contain an index parameter, which references a specific translator in the DLL that must operate on a file.

Creating a Translator DLL

You can create custom sequence file translators in any development environment that can create a Windows DLL with the required C callback functions. National Instruments recommends using the translator examples written in LabVIEW, LabWindows/CVI, and Microsoft Visual C++ as a guide. Each example in the <TestStand Public>\Examples\SequenceFileTranslators directory includes a template project, which contains source code with empty callback functions you must export from the translator DLL. You must add the necessary code to the required callbacks to ensure that the translator properly integrates with TestStand.

Example Sequence File Translators

The LabVIEW, LabWindows/CVI, and Visual C++ example projects demonstrate how to build translator DLLs and provide guidance for developing translators. The examples illustrate two simple translators for each development environment that use the TestStand API to convert sample test descriptions in XML and ASCII text formats into TestStand sequence files. The example translators for each file format produce the same TestStand sequence file.

The sample test descriptions specify steps that perform a calculation, display the result of the calculation in a graph, compare the result with an expected value, and display a message that indicates if the test passed or failed. The translation from the example format into a sequence file

involves adding steps and local variables to a sequence in a new sequence file object and configuring the steps to perform the required operations. The translators also use a custom step type TestStand loads from a type palette file that you must place in the <TestStand Public>\Components\TypePalettes directory.

Complete the following steps to use an example.

1. Open the TextTranslator or XMLTranslator directory for one of the examples in the <TestStand Public>\Examples\SequenceFileTranslators directory.
2. Copy the type NI_ExampleTranslatorTypes.ini file from the <TestStand Public>\Examples\SequenceFileTranslators directory to the <TestStand Public>\Components\TypePalettes directory.
3. Open and study the project in the development environment for the example.
4. If you make any changes to the project, rebuild the project to update the translator DLL. Copy the translator DLL into the <TestStand Public>\Components\Translators directory.
5. Launch the TestStand Sequence Editor or a TestStand User Interface to load the translator DLLs.
6. Select **File»Open** and select SampleTestFile.xml, SampleTestFile.lvtcf for the text version of the file for LabVIEW, SampleTestFile.cvltcf for the text version of the file for LabWindows/CVI, or SampleTestFile.vctcf for the text version of the file for Visual Studio C++.
7. Review the translated sequence file.
8. Launch an execution using the MainSequence in the sequence file.

Versioning Translators and Custom Sequence Files

When you edit a custom sequence file format, you can increment the version number for the file format and the content of the sequence. The file format version number identifies the structure and syntax of the file. The file version number identifies the revision of the content of the file.

If the content of a custom sequence file includes a file format version number, a translator can read files with the current file format and files with an earlier file format, and the translator can identify newer file formats it does not support. When a translator callback accesses the content of the file, the translator ensures that it can support the file format version.

For example, the CanTranslate callback uses the version number to determine if the translator can load the file. In addition, TestStand displays the return value from the GetFileFormatVersion callback in reports the Workspace Documentation tool creates.

If the content of a custom sequence file includes a file version number or revision, implement the translator to assign the version to the `PropertyObjectFile.Version` property in the `TranslateSequenceFile` callback and return the version in the `GetFileVersion` callback to ensure that the Sequence File Properties dialog box displays the file version number and the Sequence File Documentation and Workspace Documentation tools display the file version number in reports you create.

If the file formats between version numbers differ significantly, consider creating two translators in a single DLL or a separate translator in two DLLs to simplify the code necessary to translate each file format. If files contain header fields that identify the file format and the CanTranslate callback uses these fields, make sure that using two translators does not affect the performance of opening files in TestStand.

Deploying Translators and Custom Sequence Files

If you place the custom sequence file translator DLL and its support files in the `<TestStand Public>\Components\Translators` directory on the computer you use to build the deployment, the deployment utility automatically includes these files in the deployment when you enable the Deploy Files in TestStand Public Directories option on the System Source tab of the TestStand Deployment Utility. Alternatively, you can add the translator files to the workspace and set the target destination directory for the files to the `<TestStand Public>\Components\Translators` directory.

When the deployment utility analyzes a TestStand sequence file, the utility locates the code modules the steps in the sequence file call, adds the code module files to the deployment, and changes absolute path references in sequence files to relative path references to ensure that TestStand can locate the code module on the computer where you deploy the files.

You can add custom sequence files to the workspace the deployment utility uses to build a deployment. The deployment utility must load and translate custom sequence files to locate the code modules the steps in the sequence file call. However, the utility does not modify the paths in the custom sequence file and returns a warning if the utility cannot ensure that TestStand can locate the code module on the computer where you deploy

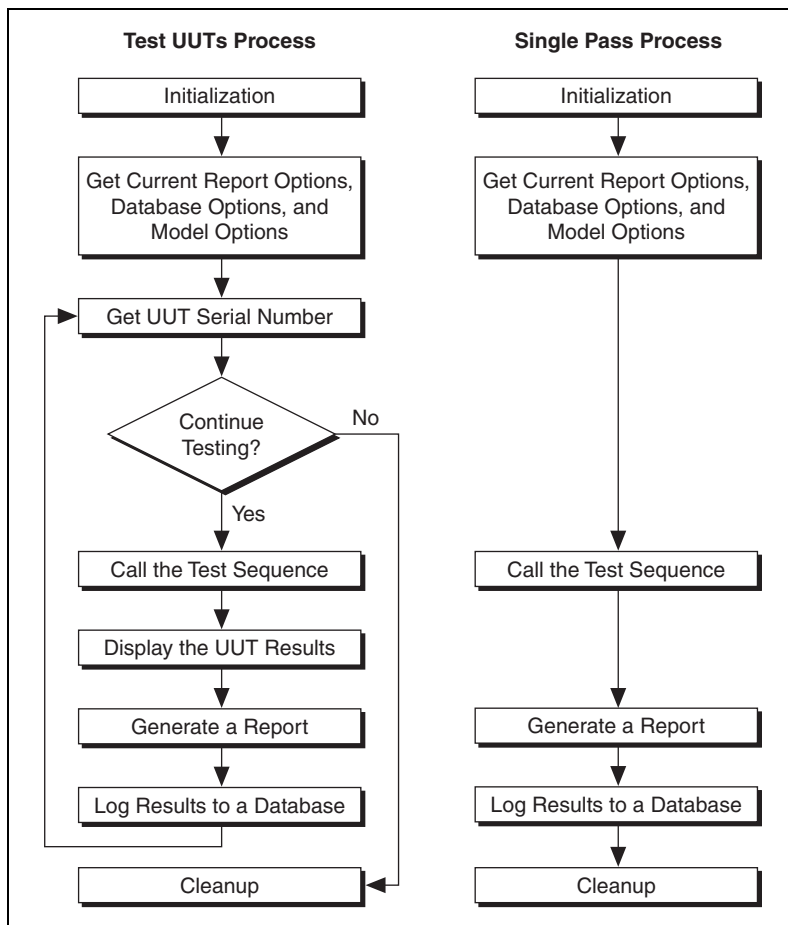
the files. You must fix the paths on the computer you use to build the deployment, or you must fix the paths on the target computer after deployment.

Refer to Chapter 14, *Deploying TestStand Systems*, and the *NI TestStand Help* for more information about creating and deploying TestStand files using the deployment utility.

Process Model Architecture

To better understand the information in this appendix, review the *Process Models* section of Chapter 1, *NI TestStand Architecture*, which includes general information about process models, entry points, and the relationship between a process model and a client sequence file.

The Sequential, Parallel, and Batch process models use the same basic structure for running a test sequence. Using the Test UUTs or Single Pass execution entry point, the process models run test sequences, generate reports, and log UUT results to a database according to configuration settings, as shown in Figure A-1.

**Figure A-1.** Process Flow

The main differences between the process models are the number of UUTs each process model runs for the Test UUTs or Single Pass Execution entry points and the way each process model relates to and synchronizes with UUTs.

TestStand Process Models

The Sequential model is the default TestStand process model. The Parallel and Batch models include features to help you implement test stations for testing multiple UUTs at the same time.

Table A-1 lists the TestStand process models and their respective sequence files.

Table A-1. TestStand Process Models

Process Model	Process Model Sequence File
Sequential Model	<TestStand>\Components\Models\TestStandModels\SequentialModel.seq
Parallel Model	<TestStand>\Components\Models\TestStandModels\ParallelModel.seq
Batch Model	<TestStand>\Components\Models\TestStandModels\BatchModel.seq

You can create your own process models, or you can modify a copy of the default TestStand process models.

To modify the installed process models or to create a new process model, copy the process model files from the <TestStand>\Components\Models\TestStandModels directory to the <TestStand Public>\Components\Models\TestStandModels directory and make changes to the copy. When you copy installed files to modify, 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 installations of the same version of TestStand do not overwrite the customizations and ensures that uninstalling TestStand does not remove the files you customize.

The list of search paths includes the subdirectories in the <TestStand Public>\Components directory, which also acts as a temporary location for components you use to build a deployment. Refer to the [Search Paths](#) section of Chapter 5, [Module Adapters](#), for more information about TestStand search directories. Refer to the [TestStand Directory Structure](#)

section of Chapter 8, *Customizing and Configuring TestStand*, for more information about TestStand directories.

When you create a custom process model, use the Model tab of the Station Options dialog box to set the custom process model sequence file as the process model for the station.

Features Common to all TestStand Process Models

All TestStand process models identify UUTs, generate test reports, log results to databases, and display UUT status information. You can use client sequence files to customize various process model operations by overriding model-defined callback sequences.

In addition to using a primary, or parent, process model file, you can use a secondary, or child, process model file to encapsulate specific functionality, such as report generation.

Process models provide Configuration and Execution entry points for configuring model settings and running client files under the model. The Configure and Execute menus of an application typically include the model entry points.

TestStand process models include the following Execution entry points:

- **Test UUTs**—Tests and identifies multiple UUTs or batches of UUTs in a loop.
- **Single Pass**—Tests one UUT or a single batch of UUTs without identifying the UUTs.



Note When you select the Test UUTs Execution entry point to start an execution that continuously tests UUTs, any subsequent configuration changes you make to the Report, Database, or Model Options entry points do not affect UUTs tested in the execution.

TestStand process models include the following Configuration entry points:

- **Report Options**—Launches the Report Options dialog box, in which you enable UUT report generation and configure the report type and content of the report files.
- **Database Options**—Launches the Database Options dialog box, in which you enable UUT result logging and configure the schema for mapping TestStand results to database tables and columns.

- **Model Options**—Launches the Model Options dialog box, in which you configure the number of test sockets and other process model-related options.

Refer to the *NI TestStand Help* for more information about the Report Options, Database Options, and Model Options dialog boxes.

Sequential Model

Use the Sequential process model to test one UUT at a time.

Parallel and Batch Models

Use the Parallel and Batch process models to simultaneously run the same test sequence on groups of similar UUTs. Select **Configure»Model Options** to launch the Model Options dialog box to specify the number of test sockets in the test system.

Parallel Model

Use the Parallel model to control multiple independent test sockets. With the Parallel model, you can start and stop testing on any test socket at any time. For example, if you have five test sockets for testing radios, you can load a new radio into an open test socket while the other test sockets continue testing other radios.

When you select the Single Pass Execution entry point, the Parallel model launches a separate execution for each test socket without prompting for UUT serial numbers.

Batch Model

Use the Batch model to control a set of test sockets that test multiple UUTs as a group. For example, if you have a set of circuit boards attached to a common carrier, use the Batch model to ensure that you start and finish testing all boards at the same time. With the synchronization features of the Batch model, you can specify if a step that applies to the batch as a whole runs only once per batch instead of once for each UUT. You can also specify if certain steps or groups of steps cannot run on more than one UUT at a time or if certain steps must run on all UUTs at the same time. The Batch model generates batch reports that summarize the test results for the UUTs in the batch.

When you select the Single Pass Execution entry point, the Batch model launches a separate execution for each test socket without prompting for UUT serial numbers.

Selecting the Default Process Model

Select **Configure»Station Options** and click the **Model** tab to change the default process model. Select a model from the Station Model ring control or click **Browse** to select a process model sequence file. You can also use the Sequence File Properties dialog box to specify that a sequence file always uses a particular process model.

Sequential Process Model

The Sequential process model, `SequentialModel.seq`, includes sequences for Execution entry points, Configuration entry points, Model callbacks, Utility sequences, and Engine callbacks. The Execution entry points are Test UUTs and Single Pass.

Execution Entry Points

The Sequential process model includes the following Execution entry point sequences:

- **Test UUTs**—Tests and identifies multiple UUTs in a loop. The Execute menu includes the Test UUTs item when a window for a client sequence file is active. Refer to the [Test UUTs](#) section of the *Sequential Process Model* section of this appendix for more information about the Test UUTs Execution entry point.
- **Single Pass**—Tests one UUT without identifying the UUTs. The Single Pass Execution entry point performs a single iteration of the loop the Test UUTs Execution entry point performs. The Execute menu includes the Single Pass item when a window for a client sequence file is active. Refer to the [Single Pass](#) section of the *Sequential Process Model* section of this appendix for more information about the Single Pass Execution entry point.

Configuration Entry Points

The Sequential process model includes the following Configuration entry point sequences:

- **Configure Report Options**—Launches the Report Options dialog box, in which you enable UUT report generation and configure the report type and content of report files. Refer to Chapter 6, [Database](#)

[Logging and Report Generation](#), for more information about report options.

- **Configure Database Options**—Launches the Database Options dialog box, in which you enable UUT result logging and configure the schema for mapping TestStand results to database tables and columns. Refer to Chapter 6, [Database Logging and Report Generation](#), for more information about database options
- **Configure Model Options**—Launches the Model Options dialog box, in which you configure the number of test sockets and other process model-related options.

The Configuration entry points save the station report, database, and model options to disk. The settings in the Report Options, Database Options, and Model Options dialog boxes apply to the test station as a whole.

Model Callbacks

The Sequential process model includes the following Model callback sequences, which you can override with client sequence files:

- **MainSequence**—Test UUTs and Single Pass Execution entry point sequences call the MainSequence callback, which is empty in the process model file. The client sequence file must contain a MainSequence callback that performs the tests on a UUT.
- **PreUUT**—Launches the UUT Information dialog box to obtain the UUT serial number. The Test UUTs Execution entry point calls the PreUUT callback at the beginning of each iteration of the UUT loop. If the operator enters a UUT serial number, the IdentifyUUT step stores the serial number in the UUT.SerialNumber parameter, which is a local variable the Test UUTs sequence passes to the PreUUT callback sequence. If the operator stops testing, the UUT loop terminates, and the IdentifyUUT step sets the ContinueTesting parameter to `False`. The ContinueTesting parameter is another local variable the Test UUTs sequence passes to the PreUUT callback sequence.
- **PostUUT**—Displays a pass, fail, error, or terminate banner to indicate the status of the test the MainSequence callback in the client sequence file performs on the UUT. The Test UUTs Execution entry point calls the PostUUT callback at the end of each iteration of the UUT loop.
- **PreUUTLoop**—Before the UUT loop begins, the Test UUTs Execution entry point calls the PreUUTLoop callback, which is empty in the process model file.

- **PostUUTLoop**—After the UUT loop terminates, the Test UUTs Execution entry point calls the PostUUTLoop callback, which is empty in the process model file.
- **ReportOptions**—After reading the test station report options from disk, the Get Report Options subsequence of the Execution entry point sequence calls the ReportOptions callback so the client sequence file can modify the report options. The ReportOptions callback in the process model file is empty.
- **DatabaseOptions**—After reading the test station database options from disk, the Get Database Options subsequence of the Execution entry point sequence calls the DatabaseOptions callback so the client sequence file can modify the database options. The DatabaseOptions callback in the process model file is empty.
- **ModelOptions**—After reading the test station model options from disk, the Get Model Options subsequence of the Execution entry point sequence calls the ModelOptions callback so the client sequence file can modify the model options. The ModelOptions callback in the process model file is empty.
- **TestReport**—The Test UUTs and Single Pass Execution entry points call the TestReport callback to generate the content of the test report for one UUT. Execution entry points do not call the TestReport callback if you enabled the On-The-Fly Reporting option in the Report Options dialog box. The Sequential process model defines a test report for a single UUT as a header, an entry for each step result, and a footer. If you do not want to override the entire TestReport callback, you can override the ModifyReportHeader, ModifyReportEntry, and ModifyReportFooter callbacks instead to customize the test report.

Based on the settings in the Report Options dialog box, the TestReport callback determines if TestStand uses sequences or a DLL to build the report body. Select the Sequence option to more easily modify the reports TestStand generates. Select the DLL option to generate reports more efficiently.

If you select the Sequence option in the Report Options dialog box, the TestReport callback calls the AddReportBody sequence in `reportgen_html.seq` or `reportgen_txt.seq` to build the report body. The report generator uses a series of sequences with steps that recursively process the result list for the execution. If you select the DLL option in the Report Options dialog box, the TestReport callback calls a function in `modelsupport2.dll` to build the report body. You can access the project and source code for the DLL built in LabWindows/CVI from the `<TestStand>\Components\Models\TestStandModels` directory.

For XML reports, the `AddReportBody` sequence in `reportgen_xml.seq` calls the `TestStand API PropertyObject.GetXML` method. For ATML reports, the `GetATMLReport` sequence in `reportgen_atml.seq` calls the `Get_Atml_Report` function in `ATML_Report.c` in the `ATMLSupport.prj` LabWindows/CVI project.

Refer to the [Report Generation Functions and Sequences](#) section of this appendix for more information about how TestStand generates reports.

- **ModifyReportHeader**—The `TestReport` callback calls the `ModifyReportHeader` callback so the client sequence file can modify the report header. The `ModifyReportHeader` callback receives parameters for the UUT information, the tentative report header text, and the report options. The `ModifyReportHeader` callback in the process model file is empty.
- **ModifyReportEntry**—The `TestReport` callback uses subsequences to call the `ModifyReportEntry` callback for each result in the result list for the UUT so the client sequence file can modify the entry point for each step result. The `ModifyReportEntry` callback receives parameters for an entry from the result list, the UUT information, the tentative report entry text, the report options, and a number that indicates the call stack depth at the time the step executed. TestStand does not call `ModifyReportEntry` callbacks if you enabled the DLL option in the Report Options dialog box. Instead, you must modify `modelsupport2.dll`, located in the `<TestStand>\Components\Models\TestStandModels` directory, to modify how step results appear in the report. The `ModifyReportEntry` callback in the process model file is empty.
- **ModifyReportFooter**—The `TestReport` callback calls the `ModifyReportFooter` callback so the client sequence file can modify the report footer. The `ModifyReportFooter` callback receives parameters for the UUT information, the tentative report footer text, and the report options. The `ModifyReportFooter` callback in the process model file is empty.
- **LogToDatabase**—Execution entry points call the `LogToDatabase` callback to populate a database with the results for one UUT. Execution entry points do not call the `LogToDatabase` callback if you enabled the Use On-The-Fly Logging option in the Database Options dialog box. The `LogToDatabase` callback receives parameters for the UUT information, the result list for the UUT, and the database options.

- **ProcessSetup**—The Test UUTs and Single Pass Execution entry points call the ProcessSetup callback from the Setup step group so the client sequence file can execute any setup steps that must run only once during the execution of the process model. The Process Setup callback in the process model file is empty.
- **ProcessCleanup**—The Test UUTs and Single Pass Execution entry points call the ProcessCleanup callback from the Cleanup step group so the client sequence file can execute any cleanup steps that must run only once during the execution of the process model. The Process Cleanup callback in the process model file is empty.

Utility Sequences

The Sequential process model sequences call the following Utility sequences:

- **Get Report Options**—Execution entry points call the Get Report Options sequence at the beginning of an execution. The Get Report Options callback reads the test station report options from disk and calls the ReportOptions callback so the client sequence file can modify the report options.
- **Get Station Info**—Execution entry points call the Get Station Info sequence at the beginning of an execution to identify the test station name and the current user.
- **Get Database Options**—Execution entry points call the Get Database Options sequence at the beginning of an execution. The Get Database Options callback reads the test station database options from disk and calls the DatabaseOptions callback so the client sequence file can modify the database options.
- **Get Model Options**—Execution entry points call the Get Model Options sequence at the beginning of an execution. The Get Model Options callback reads the test station model options from disk and calls the ModelOptions callback so the client sequence file can modify the model options.

Engine Callbacks

The Sequential process model includes the following Engine callbacks:

- **ProcessModelPostResultListEntry**—The process model enables this callback if you enable the On-The-Fly Reporting option in the Report Options dialog box or if you enable the Use On-The-Fly Logging option in the Database Options dialog box. TestStand calls this Engine callback after each step that tests a UUT and generates a step result.

- **SequenceFilePostResultListEntry**—The process model enables this callback if you enable the On-The-Fly Reporting option in the Report Options dialog box or if you enable the Use On-The-Fly Logging option in the Database Options dialog box. TestStand calls this Engine callback after any step in the process model generates a step result. However, this callback processes only results the MainSequence model callback steps generate.

Test UUTs

Open `SequentialModel.seq` in the sequence editor and select the **Test UUTs** sequence on the Sequences pane to examine the Sequential process model Test UUTs Execution entry point, which performs the following significant actions:

1. Calls the ProcessSetup callback.
2. Calls the PreUUTLoop callback.
3. Calls the Get Model Options utility sequence.
4. Calls the Get Station Info utility sequence.
5. Calls the Get Report Options utility sequence.
6. Calls the Get Database Options utility sequence.
7. Calls the Configure Post Result Callbacks utility sequence to enable the ProcessModelPostResultListEntry and SequenceFilePostResultListEntry callbacks if you enable on-the-fly report generation or database logging.
8. Increments the UUT index.
9. Calls the PreUUT callback.
10. If no more UUTs exist, skips to step 20.
11. Sets up the report by determining the report file pathname, setting up display settings, resetting the report, and setting the report location.
12. Clears information from the previous loop iteration by discarding the previous results and clearing the report and failure stacks.
13. Starts on-the-fly report generation and database logging, if enabled, for the new UUT.
14. Calls the MainSequence callback.
15. Calls the PostUUT callback.
16. Calls the TestReport callback.

17. Writes the UUT report to disk by appending to an existing file or creating a new file. Also adjusts the root tags if the report format is XML.
18. Calls the LogToDatabase callback.
19. Loops back to step 8.
20. Calls the PostUUTLoop callback.
21. Calls the ProcessCleanup callback.

Single Pass

Open `SequentialModel.seq` in the sequence editor and select the **Single Pass** sequence on the Sequences pane to examine the Sequential process model Single Pass Execution entry point, which performs the following significant actions:

1. Calls the ProcessSetup callback.
2. Calls the Get Model Options utility sequence.
3. Calls the Get Station Info utility sequence.
4. Calls the Get Report Options utility sequence.
5. Calls the Get Database Options utility sequence.
6. Calls the Configure Post Result Callbacks utility sequence to enable the `ProcessModelPostResultListEntry` and `SequenceFilePostResultListEntry` callbacks if you enable on-the-fly report generation or database logging.
7. Sets up the report by determining the report file pathname, setting up display settings, resetting the report, and setting the report location.
8. Starts on-the-fly report generation and database logging, if enabled, for the new UUT.
9. Calls the MainSequence callback.
10. Calls the TestReport callback.
11. Writes the UUT report to disk by appending to an existing file or creating a new file. Also adjusts the root tags if the report format is XML.
12. Calls the LogToDatabase callback.
13. Calls the ProcessCleanup callback.

Parallel Process Model

The Parallel process model, `ParallelModel.seq`, includes sequences for main Execution entry points, Utility sequences, hidden Execution entry points, Configuration entry points, Model callbacks, and Engine callbacks. The main Execution entry points are Test UUTs and Single Pass. The hidden Execution entry points are Test UUTs – Test Socket Entry Point and Single Pass – Test Socket Entry Point.

Main Execution Entry Points

The Parallel process model includes the following main Execution entry point sequences:

- **Test UUTs**—Initiates a hidden execution that controls the test socket executions it creates using the Test UUTs – Test Socket Entry Point sequence. The Execute menu includes the Test UUTs item when a window for a client sequence file is active. Refer to the [Test UUTs](#) section of the *Parallel Process Model* section of this appendix for more information about the Test UUTs Execution entry point.
- **Single Pass**—Initiates a hidden execution that controls the test socket executions it creates using the Single Pass – Test Socket Entry Point sequence. The Execute menu includes the Single Pass item when a window for a client sequence file is active. Refer to the [Single Pass](#) section of the *Parallel Process Model* section of this appendix for more information about the Single Pass Execution entry point.

Utility Sequences

The Parallel process model calls the same Get Report Options, Get Station Info, Get Database Options, and Get Model Options Utility sequences as the Sequential process model. Refer to the [Utility Sequences](#) section of the *Sequential Process Model* section of this appendix for information about these sequences.

The main Execution entry points in the Parallel process model use the following additional utility sequences:

- **Initialize TestSocket**—The controlling execution calls the Initialize TestSocket sequence to initialize the data for and create the test socket executions.
- **Tile Execution Windows**—The controlling execution calls the Tile Execution Windows sequence to tile the test socket Execution windows by building a list of executions and posting a `UIMessage` to the user interface that requests window tiling.

- **Monitor Threads**—The `ProcessDialogRequests` sequence calls the `Monitor Threads` sequence periodically from the controlling execution to determine if any of the test socket executions terminated or aborted. The `Monitor Threads` sequence updates the `ModelData` for terminated or aborted test sockets to indicate the new state and updates the UUT Information dialog box the controlling execution launches for that test socket.
- **ProcessDialogRequests**—The controlling execution calls the `ProcessDialogRequests` sequence from the Test UUTs sequence after displaying the UUT Information dialog box, which enqueues requests for sequence names into `ModelData.DialogRequestQueue`. The `ProcessDialogRequests` sequence loops while waiting for those requests. When the `ProcessDialogRequests` sequence receives a request, it calls the requested sequence. Additionally, the `ProcessDialogRequests` sequence periodically calls the `Monitor Threads` sequence to verify the state of and update the information for the test socket executions.
- **Run UUT Info Dialog**—The controlling execution calls the `Run UUT Info Dialog` sequence from a new thread in the Test UUTs Execution entry point to initialize and launches the UUT Information dialog box the Test UUTs Execution entry point uses to display information and gather serial numbers for the test socket executions.
- **Continue TestSocket**—The `ProcessDialogRequests` sequence calls the `Continue TestSocket` callback to notify the test socket to continue executing. The test socket execution waits on the notification in the default implementation of the `PreUUT` and `PostUUT` callbacks.
- **Terminate TestSocket**—The `ProcessDialogRequests` sequence calls this dialog box request callback. The `Terminate TestSocket` sequence terminates the execution for the test socket the request specifies.
- **Abort TestSocket**—The `ProcessDialogRequests` sequence calls this dialog box request callback. The `Abort TestSocket` sequence aborts the execution for the test socket the request specifies.
- **Restart TestSocket**—The `ProcessDialogRequests` sequence calls this dialog box request callback. The `Restart TestSocket` sequence restarts the execution for the test socket the request specifies and re-tiles the Execution windows to include the Execution window the `Restart TestSocket` sequence restarts.
- **Terminate All TestSockets**—The `ProcessDialogRequests` sequence calls this dialog box request callback. The `Terminate All TestSockets` sequence terminates all the test socket executions.

- **Abort All TestSockets**—The ProcessDialogRequests sequence calls this dialog box request callback. The Abort All TestSockets sequence aborts all the test socket executions.
- **Stop All TestSockets**—The ProcessDialogRequests sequence calls this dialog box request callback. The Stop All TestSockets sequence sets a flag for each test socket execution to stop after completing the current UUT test sequence. The sequence also sets a notification for test socket executions to continue to that point without interruption.
- **View TestSocket Report**—The ProcessDialogRequests sequence calls this dialog box request callback to enable the View Report button in the Status dialog box. Click the **View Report** button to launch a report viewer for the report file for the test socket the request specifies.
- **View TestSocket Report – Current Only**—The ProcessDialogRequests sequence calls this dialog box request callback to enable the View Report button in the Status dialog box. Click the **View Report** button to launch a report viewer for the last report generated for the test socket the request specifies. This sequence differs from the View TestSocket Report sequence because it shows only the last report instead of the whole report file.

Hidden Execution Entry Points

The main Execution entry points of the Parallel process model use but do not display the following hidden Execution entry point sequences to initiate test socket executions:

- **Test UUTs – Test Socket Entry Point**—The controlling execution uses this entry point to create the test socket executions and implement the Test UUTs Execution entry point for the test socket executions. If you insert a step into this sequence, disable the Record Result option for the step to ensure that report generation and database logging function properly. Refer to the [Test UUTs – Test Socket Entry Point](#) section of the [Parallel Process Model](#) section of this appendix for more information about this entry point.
- **Single Pass – Test Socket Entry Point**—The controlling execution uses this entry point to create the test socket executions and implement the Single Pass Execution entry point for test socket executions. If you insert a step into this sequence, disable the Record Result option for the step to ensure that report generation and database logging function properly. Refer to the [Single Pass – Test Socket Entry Point](#) section of the [Parallel Process Model](#) section of this appendix for more information about this entry point.

Configuration Entry Points

The Parallel process model includes the same Configuration entry point sequences as the Sequential process model. Refer to the [Configuration Entry Points](#) section of the [Sequential Process Model](#) section of this appendix for information about these sequences.

Model Callbacks

The Parallel process model includes the following Model callback sequences, which you can override with client sequence files:

- **MainSequence**—The Test UUTs – Test Socket Entry Point and Single Pass – Test Socket Entry Point sequences call the MainSequence callback, which is empty in the process model file. The client sequence file must contain a MainSequence callback that performs the tests on a UUT.
- **PreUUT**—Launches the UUT Information dialog box to obtain the UUT serial numbers for the test sockets. The Test UUTs – Test Socket Entry Point sequence calls the PreUUT callback at the beginning of each iteration of the UUT loop. If the operator enters a serial number, the code for the dialog box stores the serial number in the TestSocket.UUT.SerialNumber parameter. If the operator stops testing, the UUT loop terminates, and the code for the dialog box sets the TestSocket.ContinueTesting parameter to `False`.
- **PostUUT**—Displays a pass, fail, error, or terminate banner to indicate the status of the test the MainSequence callback in the client sequence file performs on the UUT. The Test UUTs – Test Socket Entry Point sequence calls the PostUUT callback at the end of each iteration of the UUT loop.
- **PreUUTLoop**—Before the UUT loop begins, the Test UUTs – Test Socket Entry Point sequence calls the PreUUTLoop callback, which is empty in the process model file.
- **PostUUTLoop**—After the UUT loop terminates, the Test UUTs – Test Socket Entry Point sequence calls the PostUUTLoop callback, which is empty in the process model file.
- **ReportOptions, DatabaseOptions, ModelOptions, TestReport, ModifyReportHeader, ModifyReportEntry, ModifyReportFooter, and LogToDatabase**—Refer to the [Model Callbacks](#) section of the [Sequential Process Model](#) section of this appendix for information about these sequences.
- **ProcessSetup**—The Test UUTs and Single Pass Execution entry points call the ProcessSetup callback from the Setup step group so the

client sequence file can execute any setup steps that must run only once during the execution of the process model. Only the controlling execution runs these setup steps. The test socket executions do not call the ProcessSetup callback.

- **ProcessCleanup**—The Test UUTs and Single Pass Execution entry points call the ProcessCleanup callback from the Cleanup step group so the client sequence file can execute any cleanup steps that must run only once during the execution of the process model. Only the controlling execution runs these cleanup steps. The test socket executions do not call the ProcessCleanup callback.

Engine Callbacks

The Parallel process model includes the same Engine callbacks as the Sequential process model. Refer to the [Engine Callbacks](#) section of the [Sequential Process Model](#) section of this appendix for information about these sequences.

Test UUTs

The Test UUTs Execution entry point is the sequence the controlling execution runs.

Open `ParallelModel.seq` in the sequence editor and select the **Test UUTs** sequence on the Sequences pane to examine the Parallel process model Test UUTs Execution entry point, which performs the following significant actions:

1. Calls the ProcessSetup callback.
2. Calls the Get Model Options utility sequence.
3. Calls the Get Station Info utility sequence.
4. Calls the Get Report Options utility sequence.
5. Calls the Get Database Options utility sequence.
6. Calls the Run UUT Info Dialog utility sequence.
7. Determines the report file pathname to use when you configure the report options to write all UUT results for the model to the same file.
8. Creates and initializes the test socket executions. Refer to the [Test UUTs – Test Socket Entry Point](#) section of the [Parallel Process Model](#) section of this appendix for more information about the sequence file the test socket executions run.
9. Calls the ProcessDialogRequests utility sequence.
10. Calls the ProcessCleanup callback.

Test UUTs – Test Socket Entry Point

The Test UUTs – Test Socket entry point is the sequence the test socket executions run. The controlling execution creates the test socket executions in the Test UUTs Execution entry point sequence.

Open `ParallelModel.seq` in the sequence editor and select the **Test UUTs – Test Socket Entry Point** sequence on the Sequences pane to examine the Parallel process model Test UUTs – Test Socket entry point, which performs the following significant actions:

1. Calls the `PretUUTLoop` callback.
2. Calls the `Configure Post Result Callbacks` utility sequence to enable the `ProcessModelPostResultListEntry` and `SequenceFilePostResultListEntry` callbacks if you enable on-the-fly report generation or database logging.
3. Increments the UUT index.
4. Clears information from the previous loop iteration by discarding the previous results and clearing the report and failure stacks.
5. Calls the `PreUUT` callback.
6. If no more UUTs exist, skips to step 15.
7. Sets up the report by determining the report file pathname, setting up display settings, resetting the report, and setting the report location.
8. Starts on-the-fly report generation and database logging, if enabled, for the new UUT.
9. Calls the `MainSequence` callback.
10. Calls the `PostUUT` callback.
11. Calls the `TestReport` callback.
12. Calls the `LogToDatabase` callback.
13. Writes the UUT report to disk by appending to an existing file or creating a new file. Also adjusts the root tags if the report format is XML.
14. Loops back to step 3.
15. Calls the `PostUUTLoop` callback.

Single Pass

The Single Pass Execution entry point is the sequence the controlling execution runs.

Open `ParallelModel.seq` in the sequence editor and select the **Single Pass** sequence on the Sequences pane to examine the Parallel process model Single Pass Execution entry point, which performs the following significant actions:

1. Calls the ProcessSetup callback.
2. Calls the Get Model Options utility sequence.
3. Calls the Get Station Info utility sequence.
4. Calls the Get Report Options utility sequence.
5. Calls the Get Database Options utility sequence.
6. Determines the report file pathname to use when you configure the report options to write all UUT results for the model to the same file.
7. Creates and initializes the test socket executions. Refer to the *Single Pass – Test Socket Entry Point* section of the [Parallel Process Model](#) section of this appendix for more information about the sequence file the test socket executions run.
8. Waits for test socket executions to complete.
9. Calls the ProcessCleanup callback.

Single Pass – Test Socket Entry Point

The Single Pass – Test Socket entry point is the sequence the test socket executions run. The controlling execution creates the test socket executions in the Single Pass Execution entry point sequence.

Open `ParallelModel.seq` in the sequence editor and select the **Single Pass – Test Socket Entry Point** sequence on the Sequences pane to examine the Parallel process model Single Pass – Test Socket entry point, which performs the following significant actions:

1. Calls the Configure Post Result Callbacks utility sequence to enable the `ProcessModelPostResultListEntry` and `SequenceFilePostResultListEntry` callbacks if you enable on-the-fly report generation or database logging.
2. Sets up the report by determining the report file pathname, setting up display settings, resetting the report, and setting the report location.
3. Starts on-the-fly report generation and database logging, if enabled, for the new UUT.

4. Calls the MainSequence callback.
5. Calls the TestReport callback.
6. Calls the LogToDatabase callback.
7. Writes the UUT report to disk by appending to an existing file or creating a new file. Also adjusts the root tags if the report format is XML.

Batch Process Model

The Batch process model, `BatchModel.seq`, includes sequences for main Execution entry points, Utility sequences, hidden Execution entry points, Configuration entry points, Model callbacks, and Engine callbacks. The main Execution entry points are Test UUTs and Single Pass. The hidden Execution entry points are Test UUTs – Test Socket Entry Point and Single Pass – Test Socket Entry Point.

Main Execution Entry Points

The Batch process model includes the following main Execution entry point sequences:

- **Test UUTs**—Initiates an execution that controls a separate execution for each test socket using the Test UUTs – Test Socket Entry Point sequence. The Test UUTs sequence adds the main threads of those executions to a Batch Synchronization object and controls the flow of execution using queues and notifications so all test socket executions execute the Main sequence of the client sequence file together as a group. After a group of UUTs executes, the Test UUTs sequence generates a batch report, loops back to run the client sequence file on the next group of UUTs, and controls the subsidiary test socket executions to keep them synchronized. The Execute menu includes the Test UUTs item when a window for a client sequence file is active. Refer to the [Test UUTs](#) section of the *Batch Process Model* section of this appendix for more information about the Test UUTs Execution entry point.
- **Single Pass**—Initiates an execution that controls a separate execution for each test socket using the Single Pass – Test Socket Entry Point sequence. The Single Pass sequence adds the main threads of those executions to a Batch Synchronization object and controls the flow of execution using queues and notifications so all test socket executions execute the Main sequence of the client sequence file together as a group. After the group of UUTs executes, the Single Pass sequence generates a batch report and waits for all subsidiary executions to

complete. The Execute menu includes the Single Pass item when a window for a client sequence file is active. Refer to the [Single Pass](#) section of the [Batch Process Model](#) section of this appendix for more information about the Single Pass Execution entry point.

Utility Sequences

The Batch process model calls the same Get Report Options, Get Station Info, Get Database Options, and Get Model Options Utility sequences as the Sequential process model. Refer to the [Utility Sequences](#) section of the [Sequential Process Model](#) section of this appendix for information about these sequences.

The main Execution entry points in the Batch process model use the following additional Utility sequences:

- **Restart TestSocket**—The ProcessDialogRequests sequence calls the Restart TestSocket callback to restart the execution for the test socket the request specifies.
- **Initialize TestSocket**—The controlling execution calls the Initialize TestSocket sequence to initialize the data for and create the test socket executions.
- **Monitor Batch Threads**—The ProcessDialogRequests, ProcessTestSocketRequests, and WaitForTestSocket sequences call the Monitor Batch Threads sequence periodically from the controlling execution to determine if any of the test socket executions terminated or aborted. The Monitor Batch Threads sequence updates the ModelData parameter for terminated or aborted test sockets to indicate the new state and updates the UUT Information dialog box the controlling execution launches for that test socket.
- **Tile Execution Windows**—The controlling execution calls the Tile Execution Windows sequence to tile the test socket Execution windows by building a list of executions and posting a UIMessage to the user interface that requests window tiling. The Tile Execution Windows sequence tiles only running, non-disabled test socket executions.
- **Add TestSocket Threads to Batch**—The Test UUTs and Single Pass Execution entry points call the Add TestSocket Threads to Batch sequence from the controlling execution to add the main threads of the test socket executions to a Batch Synchronization object. The threads remove themselves from the batch in the Test UUTs – Test Socket Entry Point and the Single Pass – Test Socket Entry Point sequences after running the Main sequence of the client sequence file to clean up

the state of the batch in case the sequence terminates or the client sequence file did not properly handle batch synchronization.

- **Notify TestSocket Threads**—The controlling execution calls the `Notify TestSocket Threads` sequence to make the running test socket execution threads continue executing after waiting at the last call to the `SendControllerRequest` sequence. The `Notify TestSocket Threads` sequence optionally waits for each test socket to reach the `SendControllerRequest` sequence, which serializes the execution of each test socket.
- **All TestSockets Waiting?**—Returns `True` if all running test sockets are waiting for the `WaitingForRequest` parameter or if all test sockets stop.
- **ProcessTestSocketRequests**—The controlling execution calls the `ProcessTestSocketRequests` sequence to wait for the test socket executions to synchronize at a point the controlling execution defines in the process model. When all running test sockets reach this point, the `ProcessTestSocketRequests` sequence returns and allows the controlling execution to continue. While waiting for the test sockets, the `ProcessTestSocketRequests` sequence monitors the test socket threads to make sure the threads continue to run. If all test sockets stop running, the `ProcessTestSocketRequests` sequence returns to allow the controlling sequence to continue.
- **WaitForTestSocket**—The controlling execution calls the `WaitForTestSocket` sequence from the `Notify TestSocket Threads` sequence to make a test socket execution wait to receive its next controller request, such as a synchronization point, before the next test socket execution continues. Using the `WaitForTestSocket` sequence guarantees that the controlling execution allows only one test socket to run particular sections of its sequence at a time. Use the `WaitForTestSocket` sequence to write the test socket reports to a file in test socket index order when you configure the report options to write reports to the same file.
- **ProcessDialogRequests**—The controlling execution calls the `ProcessDialogRequests` sequence from the `Test UUTs` sequence after displaying the `UUT Information` dialog box, which enqueues requests for sequence names into `ModelData.DialogRequestQueue`. The `ProcessDialogRequests` sequence loops while waiting for those requests. When the `ProcessDialogRequests` sequence receives a request, it calls the requested sequence. Additionally, the `ProcessDialogRequests` sequence periodically calls the `Monitor Batch Threads` sequence to verify the state of and update the information for the test socket executions.

- **Run Batch Info Dialog**—The controlling execution calls the Run Batch Info Dialog sequence from a new thread in the Test UUTs Execution entry point to initialize and run the dialog box in which users enter serial numbers and view the results for a particular run of the batch.
- **View TestSocket Report**—The ProcessDialogRequests sequence calls this dialog box request callback to enable the View Report button in the Status dialog box. Click the **View Report** button to launch a report viewer for the report file for the test socket the request specifies.
- **View TestSocket Report – Current Only**—The ProcessDialogRequests sequence calls this dialog box request callback to enable the View Report button in the Status dialog box. Click the **View Report** button to launch a report viewer for the last report generated for the test socket the request specifies. This sequence differs from the View TestSocket Report sequence because it shows only the last report instead of the whole report file.
- **View Batch Report**—The ProcessDialogRequests sequence calls this dialog box request callback to enable the View Report button in the Status dialog box. Click the **View Report** button to launch a report viewer for the batch report file.
- **View Batch Report – Current Only**—The ProcessDialogRequests sequence calls this dialog box request callback to enable the View Report button in the Status dialog box. Click the **View Report** button to launch a report viewer for the last batch report generated. This sequence differs from the View Batch Report sequence because it shows only the last report instead of the whole batch report file.

The hidden Execution entry points in the Batch process model call the following utility sequence:

- **SendControllerRequest**—The test socket executions call the SendControllerRequest sequence to synchronize the controlling execution at various locations in the sequences. The test socket executions pass string parameters to indicate the reason and location at which the test socket executions attempt to synchronize with the other executions. When all the running test socket executions synchronize with the controlling sequence at the same location by calling the SendControllerRequest sequence, the sequence of the controlling execution performs operations and notifies the test socket execution when to continue.

Hidden Execution Entry Points

The main Execution entry points of the Batch process model use but do not display the following hidden Execution entry point sequences to initiate test socket executions:

- **Test UUTs – Test Socket Entry Point**—The controlling execution uses this entry point to create the test socket executions and implement the Test UUTs Execution entry point for the test socket executions. If you insert a step into this sequence, disable the Record Result option for the step to ensure that report generation and database logging function properly. Refer to the [Test UUTs – Test Socket Entry Point](#) section of the [Batch Process Model](#) section of this appendix for more information about this entry point.
- **Single Pass – Test Socket Entry Point**—The controlling execution uses this entry point to create the test socket executions and implement the Single Pass Execution entry point for the test socket executions. If you insert a step into this sequence, disable the Record Result option for the step to ensure that report generation and database logging function properly. Refer to the [Single Pass – Test Socket Entry Point](#) section of the [Batch Process Model](#) section of this appendix for more information about this entry point.

Configuration Entry Points

The Batch process model includes the same Configuration entry point sequences as the Sequential process model. Refer to the [Configuration Entry Points](#) section of the [Sequential Process Model](#) section of this appendix for information about these sequences.

Model Callbacks

The Batch process model includes the following Model callback sequences, which you can override with client sequence files:

- **MainSequence**—The Test UUTs – Test Socket Entry Point and Single Pass – Test Socket Entry Point sequences call the MainSequence callback, which is empty in the process model file. The client sequence file must contain a MainSequence callback that performs the tests on a UUT.
- **PreUUT**—The test socket executions call the PreUUT callback, which is empty in the process model file. If you override the PreUUT callback with a client sequence file to obtain the serial number for the UUT, override the PreBatch callback also. The PreBatch callback launches a dialog box to obtain the serial numbers for all the UUTs in

the batch. Refer to the sequence files in the <TestStand Public>\Examples\ProcessModels\BatchModel directory for examples of how to override the PreUUT and PreBatch callbacks.

- **PostUUT**—The test socket executions call the Post UUT callback, which is empty in the process model file. If you override the PostUUT callback with a client sequence file to display the result status for a UUT, override the PostBatch callback also. The PostBatch callback launches a dialog box to show the result status for all the UUTs in the batch. Refer to the sequence files in the <TestStand Public>\Examples\ProcessModels\BatchModel directory for examples of how to override the PostUUT and PostBatch callbacks.
- **PreUUTLoop**—Before the UUT loop begins, the Test UUTs – Test Socket Entry Point sequence calls the PreUUTLoop callback, which is empty in the process model file.
- **PostUUTLoop**—After the UUT loop terminates, the Test UUTs – Test Socket Entry Point sequence calls the PostUUTLoop callback, which is empty in the process model file.
- **ReportOptions, DatabaseOptions, ModelOptions, TestReport, ModifyReportHeader, ModifyReportEntry, ModifyReportFooter, and LogToDatabase**—Refer to the [Model Callbacks](#) section of the [Sequential Process Model](#) section of this appendix for more information about these sequences.
- **ProcessSetup**—The Test UUTs and Single Pass Execution entry points call the ProcessSetup callback from the Setup step group so the client sequence file can execute any setup steps that must run only once during the execution of the process model. Only the controlling execution runs these setup steps. The test socket executions do not call the ProcessSetup callback.
- **ProcessCleanup**—The Test UUTs and Single Pass Execution entry points call the ProcessCleanup callback from the Cleanup step group so the client sequence file can execute any cleanup steps that must run only once during the execution of the process model. Only the controlling execution runs these cleanup steps. The test socket executions do not call the ProcessCleanup callback.

The main Execution entry points in the Batch process model call the following Model callback sequences, which you can override with client sequence files:

- **PreBatch**—Launches a dialog box to obtain the batch and UUT serial numbers. Refer to the sequence files in the <TestStand Public>\Examples\ProcessModels\BatchModel directory for an example of how to override the PreBatch callback.

- **PostBatch**—Displays a pass, fail, error, or terminated banner and batch and UUT reports for each test socket. Refer to the sequence files in the <TestStand Public>\Examples\ProcessModels\BatchModel directory for an example of how to override the PostBatch callback.
- **PreBatchLoop**—Before looping on a batch of UUTs, the process model calls the PreBatchLoop callback, which is empty in the process model file. Use the PreBatchLoop callback to perform an action before testing the batch.
- **PostBatchLoop**—After looping on a batch of UUTs, the process model calls the PostBatchLoop callback, which is empty in the process model file. Use the PostBatchLoop callback to perform an action after testing all batches of UUTs.
- **BatchReport**—The Test UUTs and Single Pass Execution entry points call the BatchReport callback to generate the content of the batch report for the UUTs that ran in the last batch. The Batch process model defines a batch report for a single group of UUTs as a header, an entry for each UUT result, and a footer. If you do not want to override the entire BatchReport callback, you can override the ModifyBatchReportHeader, ModifyBatchReportEntry, and ModifyBatchReportFooter callbacks instead to customize the batch report.
- **ModifyBatchReportHeader**—The BatchReport callback calls the ModifyBatchReportHeader callback so the client sequence file can modify the batch report header. The ModifyBatchReportHeader callback receives parameters for the batch serial number, the tentative report header text, and the report options. The ModifyBatchReportHeader callback in the process model file is empty.
- **ModifyBatchReportEntry**—The BatchReport callback uses subsequences to call the ModifyBatchReportEntry callback for each test socket so the client sequence file can modify the entry for each UUT result for each test socket. The ModifyBatchReportEntry callback receives parameters for the test socket data, the batch serial number, the tentative report entry text, and the report options. The ModifyBatchReportEntry callback in the process model file is empty.
- **ModifyBatchReportFooter**—The BatchReport callback calls the ModifyBatchReportFooter callback so the client sequence file can modify the batch report footer. The ModifyBatchReportFooter callback receives parameters for the tentative report footer text and the report options. The ModifyBatchReportFooter callback in the process model file is empty.

Engine Callbacks

The Batch process model includes the same Engine callbacks as the Sequential process model. Refer to the [Engine Callbacks](#) section of the [Sequential Process Model](#) section of this appendix for information about these sequences.

Test UUTs

The Test UUTs Execution entry point is the sequence the controlling execution runs.

Open `BatchModel.seq` in the sequence editor and select the **Test UUTs** sequence on the Sequences pane to examine the Batch process model Test UUTs Execution entry point, which performs the following significant actions:

1. Calls the `ProcessSetup` callback.
2. Calls the `Get Model Options` utility sequence.
3. Calls the `PreBatchLoop` callback.
4. Calls the `Get Station Info` utility sequence.
5. Calls the `Get Report Options` utility sequence.
6. Calls the `Get Database Options` utility sequence.
7. Creates and initializes the test socket executions. Refer to the [Test UUTs – Test Socket Entry Point](#) section of the [Batch Process Model](#) section of this appendix for more information about the sequence file the test socket executions run.
8. Calls the `Run Batch Info Dialog` utility sequence.
9. Calls the `ProcessTestSocketRequests` utility sequence to wait for and monitor test socket executions as the test sockets synchronize before beginning initialization.
10. Calls the `Add TestSocket Threads to Batch` utility sequence.
11. Calls the `Notify TestSocket Threads` utility sequence to notify test sockets to continue initialization.
12. Increments the Batch index.
13. Calls the `ProcessTestSocketRequests` utility sequence to wait for and monitor test socket executions as the test sockets obtain UUT serial numbers.
14. Calls the `PreBatch` callback.

15. If no more UUTs exist, sets the ContinueTesting test socket data variable to `False` for all the test sockets and marks all test sockets as enabled to add them to the batch and so they exit normally.
16. Adds enabled test socket threads to the batch and removes disabled test sockets from the batch so they do not block running threads.
17. Calls the Notify TestSocket Threads utility sequence to notify test sockets to continue obtaining UUT serial numbers.
18. If no more UUTs exist, skips to step 35.
19. Calls the ProcessTestSocketRequests utility sequence to wait for and monitor test socket executions when the test sockets are ready to run.
20. Determines the report file pathname to use for the batch and UUT report files when you configure the report options to write all UUT results for the model to the same file or to the same file as the batch reports.
21. Calls the Notify TestSocket Threads utility sequence to notify test sockets to continue running.
22. Calls the ProcessTestSocketRequests utility sequence to wait for and monitor test socket executions as the test sockets display the test socket status.
23. Calls the Add TestSocket Threads to Batch utility sequence to add test socket execution threads to the batch again if necessary.
24. Calls the PostBatch callback.
25. Calls the Notify TestSocket Threads utility sequence to notify test sockets to continue displaying the test socket status.
26. Calls the ProcessTestSocketRequests utility sequence to wait for and monitor test socket executions as TestStand generates reports for the test sockets.
27. Calls the BatchReport callback.
28. Writes the batch report to disk by appending to an existing file or creating a new file. Also adjusts the root tags if the report format is XML.
29. Calls the Notify TestSocket Threads utility sequence and returns `True` for the ReleaseThreadsSequentially parameter so TestStand writes only one UUT report at a time in the test socket index order to notify test sockets to continue.
30. Calls the ProcessTestSocketRequests utility sequence to wait for and monitor test socket executions as the test sockets complete execution.

31. Notifies the Status dialog box when report generation completes and enables the View Report button so you can view the reports from the dialog box.
32. Waits for the Status dialog box. If TestStand launches the PostBatch callback Status dialog box, the sequence waits for you to dismiss the dialog box if you have not already done so.
33. Calls the Notify TestSocket Threads utility sequence to notify test sockets to continue completing execution.
34. Loops back to step 12.
35. Waits for test socket executions to complete.
36. Calls the PostBatchLoop callback.
37. Calls the ProcessCleanup callback.

Test UUTs – Test Socket Entry Point

The Test UUTs – Test Socket entry point is the sequence the test socket executions run. The controlling execution creates the test socket executions in the Test UUTs Execution entry point sequence.

Open `BatchModel.seq` in the sequence editor and select the **Test UUTs – Test Socket Entry Point** sequence on the Sequences pane to examine the Batch process model Test UUTs – Test Socket entry point, which performs the following significant actions:

1. Calls the `SendControllerRequest` utility sequence to synchronize with the controlling execution before performing initialization. The sequence waits until the controlling execution allows the test socket to perform initialization.
2. Calls the `PreUUTLoop` callback.
3. Calls the `Configure Post Result Callbacks` utility sequence to enable the `ProcessModelPostResultListEntry` and `SequenceFilePostResultListEntry` callbacks if you enable on-the-fly report generation or database logging.
4. Increments the UUT index.
5. Clears the information from the previous loop iteration by discarding the previous results and clearing the report and failure stacks.
6. Calls the `SendControllerRequest` utility sequence to synchronize with the controlling execution before retrieving UUT serial numbers. The sequence waits until the controlling execution allows the test socket to obtain UUT serial numbers.
7. Calls the `PreUUT` callback.

8. If no more UUTs exist, skips to step 22.
9. Calls the `SendControllerRequest` utility sequence to synchronize with the controlling execution before the test socket is ready to run. The sequence waits until the controlling execution allows the test socket to run.
10. Sets up the report. Determines the report file pathname, sets up the display settings, resets the report, and sets the report location.
11. Starts on-the-fly report generation and database logging, if enabled, for the new UUT.
12. Calls the `MainSequence` callback.
13. Removes the test socket thread from batch synchronization by cleaning up the state of the batch in case the Main sequence terminates or the client sequence file did not properly handle batch synchronization. The controlling execution adds the thread to batch synchronization before continuing past the next synchronization point. The controlling execution does not add disabled test sockets to the batch.
14. Calls the `SendControllerRequest` utility sequence to synchronize with the controlling execution before displaying the test socket status. The sequence waits until the controlling execution allows the test socket to display the test socket status.
15. Calls the `PostUUT` callback.
16. Calls the `TestReport` callback.
17. Calls the `LogToDatabase` callback.
18. Calls the `SendControllerRequest` utility sequence to synchronize with controlling executions before `TestStand` generates reports for the test sockets. The sequence waits until the controlling execution allows `TestStand` to generate reports.
19. Writes the UUT report to disk by appending to an existing file or creating a new file. Also adjusts the root tags if the report format is XML.
20. Calls the `SendControllerRequest` utility sequence to synchronize with controlling executions before completing execution. The sequence waits until the controlling execution allows the test socket to complete execution.
21. Loops back to step 4.
22. Calls the `PostUUTLoop` callback.

Single Pass

The Single Pass Execution entry point is the sequence the controlling execution runs.

Open `BatchModel.seq` in the sequence editor and select the **Single Pass** sequence on the Sequences pane to examine the Batch process model Single Pass Execution entry point, which performs the following significant actions:

1. Calls the `ProcessSetup` callback.
2. Calls the `Get Model Options` utility sequence.
3. Calls the `Get Station Info` utility sequence.
4. Calls the `Get Report Options` utility sequence.
5. Calls the `Get Database Options` utility sequence.
6. Creates and initializes test socket executions. Refer to the [Single Pass – Test Socket Entry Point](#) section of the *Batch Process Model* section of this appendix for more information about the sequence file the test socket executions run.
7. Calls the `ProcessTestSocketRequests` utility sequence to wait for and monitor test socket executions when the test sockets are ready to run.
8. Calls the `Add TestSocket Threads to Batch` utility sequence.
9. Determines the report file pathname to use for the batch and UUT report files when you configure the report options to write all UUT results for the model to the same file or to the same file as the batch reports.
10. Calls the `Notify TestSocket Threads` utility sequence to notify test sockets to continue running.
11. Calls the `ProcessTestSocketRequests` utility sequence to wait for and monitor test socket executions as the test sockets synchronize after executing the `MainSequence` callback.
12. Calls the `Add TestSocket Threads to Batch` utility sequence to add test socket execution threads to the batch again if necessary.
13. Calls the `Notify TestSocket Threads` utility sequence to notify test sockets to continue synchronizing after executing the `MainSequence` callback.
14. Calls the `ProcessTestSocketRequests` utility sequence to wait for and monitor test socket executions as `TestStand` generates reports for the test sockets.
15. Calls the `BatchReport` callback.

16. Writes the batch report to disk by appending to an existing file or creating a new file. Also adjusts the root tags if the report format is XML.
17. Calls the Notify TestSocket Threads utility sequence and returns `True` for the `ReleaseThreadsSequentially` parameter so TestStand writes only one UUT report at a time in test socket index order to notify test sockets to continue.
18. Calls the ProcessTestSocketRequests utility sequence to wait for and monitor test socket executions as the test sockets complete execution.
19. Calls the Notify TestSocket Threads utility sequence to notify test sockets to continue completing execution.
20. Waits for test socket executions to complete.
21. Calls the ProcessCleanup callback.

Single Pass – Test Socket Entry Point

The Single Pass – Test Socket entry point is the sequence the test socket executions run. The controlling execution creates the test socket executions in the Single Pass Execution entry point sequence.

Open `BatchModel.seq` in the sequence editor and select the **Single Pass – Test Socket Entry Point** sequence on the Sequences pane to examine the Batch process model Single Pass – Test Socket entry point, which performs the following significant actions:

1. Calls the Configure Post Result Callbacks utility sequence to enable the `ProcessModelPostResultListEntry` and `SequenceFilePostResultListEntry` callbacks if you enable on-the-fly report generation or database logging.
2. Calls the `SendControllerRequest` utility sequence to synchronize with the controlling execution before the test socket is ready to run. The sequence waits until the controlling execution allows the test socket to run.
3. Sets up the report. Determines the report file pathname, sets up the display settings, resets the report, and sets the report location.
4. Starts on-the-fly report generation and database logging, if enabled, for the new UUT.
5. Calls the `MainSequence` callback.
6. Removes the test socket thread from batch synchronization by cleaning up the state of the batch in case the `Main` sequence terminates or the client sequence file did not properly handle batch synchronization. The controlling execution adds the thread to batch synchronization before

continuing past the next synchronization point. The controlling execution does not add disabled test sockets to the batch.

7. Calls the `SendControllerRequest` utility sequence to synchronize with the controlling execution before synchronizing after executing the `MainSequence` callback. The sequence waits until the controlling execution allows the test socket to synchronize after executing the `MainSequence` callback.
8. Calls the `TestReport` callback.
9. Calls the `LogToDatabase` callback.
10. Calls the `SendControllerRequest` utility sequence to synchronize with the controlling execution before `TestStand` generates a report for the test socket. The sequence waits until the controlling execution allows `TestStand` to generate reports.
11. Writes the UUT report to disk by appending to an existing file or creating a new file. Also adjusts the root tags if the report format is XML.
12. Calls the `SendControllerRequest` utility sequence to synchronize with the controlling execution before completing execution. The sequence waits until the controlling execution allows the test socket to complete execution.

Process Model Support Files

Many sequences in the `TestStand` process model files call functions in DLLs and subsequences in other sequence files. Table A-2 lists the process model support files `TestStand` installs in the `<TestStand>\Components\Models\TestStandModels` directory.

Table A-2. Installed Support Files for the Process Model Files

File Name	Description
<code>ATMLSupport.dll</code>	DLL that contains C functions the process model sequences call to generate ATML reports.
<code>ATMLSupport.lib</code>	Import library for <code>ATMLsupport.dll</code> .
<code>banners.c</code>	C source for functions that display status banners.
<code>BatchModel.seq</code>	Entry point and Model callback sequences for the Batch process model.

Table A-2. Installed Support Files for the Process Model Files (Continued)

File Name	Description
batchuutdlg.c	C source for the functions that launch the UUT Information dialog box for the Batch process model. The modelsupport2.dll includes this file, but the default process model, SequentialModel.seq, does not call this file.
c_report.c	C source for generating HTML, XML, and ASCII text reports for the DLL option in the Select a Report Generator for Producing the Report Body section of the Contents tab of the Report Options dialog box.
ColorselectPopup.c	C source for the functions that display a dialog box in which you can select a color.
ColorSelectPopup.h	C header file that contains declarations for the function in ColorselectPopup.c.
main.c	C source for utility functions.
ModelOptions.c	C source for the functions that launch the Model Options dialog box and read and write the model options from and to disk.
modelpanels.h	C header file that contains declarations for the panels in modelpanels.uir.
modelpanels.uir	LabWindows/CVI user interface resource file that contains panels the functions in modelsupport2.dll use.
modelsupport2.dll	DLL that contains C functions the process model sequences call. Includes functions that launch the Report Options and Model Options dialog boxes, read and write those options from and to disk, determine the report file pathname, obtain the UUT serial number from the operator, and display status banners.
modelsupport2.fpc	LabWindows/CVI function panels for the functions in modelsupport2.dll.
modelsupport2.h	C header file that contains declarations for the functions in modelsupport2.dll.
modelsupport2.lib	Import library in Visual C/C++ format for modelsupport2.dll.
modelsupport2.prj	LabWindows/CVI project that builds modelsupport2.dll and modelsupport2.cws.
ModelSupport.seq	Subsequences all process models use for report generation.

Table A-2. Installed Support Files for the Process Model Files (Continued)

File Name	Description
ParallelModel.seq	Entry point and Model callback sequences for the Parallel process model.
paralleluutdlg.c	C source for the functions that launch the UUT Information dialog box for the Parallel process model. The modelsupport2.dll includes this file, but the default process model, SequentialModel.seq, does not call this file.
PropertyObject.xsd	XML schema that defines the content of the XML the PropertyObject.GetXML method generates and the PropertyObject.SetXML method requires. TestStand XML reports that Report.xsd defines also use PropertyObject.xsd.
report.c	C source for functions that launch the Report Options dialog box, read and write the report options from and to disk, and determine the report file pathname.
report.h	C header file that contains declarations for the functions in report.c.
Report.xsd	XML schema that defines the content of TestStand XML reports.
reportgen_atml.seq	Subsequences that add the header, result entries, and footer for a UUT to an ATML test report.
reportgen_html.seq	Subsequences that add the header, result entries, and footer for a UUT to an HTML test report.
reportgen_txt.seq	Subsequences that add the header, result entries, and footer for a UUT to an ASCII text test report.
reportgen_xml.seq	Subsequences that add the header, result entries, and footer for a UUT to an XML test report.
SequentialModel.seq	Entry point and Model callback sequences for the Sequential process model.
uutdlg.c	C source for the function that launches the UUT Information dialog box to obtain the UUT serial number from the operator.

You can view the content of the `reportgen_atml.seq`, `reportgen_html.seq`, `reportgen_txt.seq`, and `reportgen_xml.seq` files in the sequence editor. These files are model sequence files and contain an empty `ModifyReportEntry` callback you can override with a client sequence file. Each `reportgen` sequence file includes a `PutOneResultInReport` sequence that calls the `ModifyReportEntry` callback.

The TestStand process model sequence files also contain an empty `ModifyReportEntry` callback, even though no sequences in the process model sequence files call the `ModifyReportEntry` callback directly. The files contain a `ModifyReportEntry` callback so that the `ModifyReportEntry` callback appears in the Sequence File Callbacks dialog box for the client sequence file.

Report Generation Functions and Sequences

If you want to customize report generation for a test station, modify the default TestStand process model files or create a new process model. To modify an installed default process model or create a new process model, copy the default process model files from the `<TestStand>\Components\Models\TestStandModels` directory to the `<TestStand Public>\Components\Models\TestStandModels` directory and make changes to the copy. When you copy installed files to modify, 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.

Table A-3 lists the default process model sequences in the `<TestStand>\Components\Models\TestStandModels` directory that generate report headers and footers.

Table A-3. Sequences that Generate Report Headers and Footers

Report Format	Header	Footer
ATML	GetATMLReport sequence in reportgen_atml.seq generates the header when it generates the report body.	GetATMLReport sequence in reportgen_atml.seq generates the footer when it generates the report body.
HTML	AddReportHeader sequence in reportgen_html.seq	AddReportFooter sequence in reportgen_html.seq
Text	AddReportHeader sequence in reportgen_txt.seq	AddReportFooter sequence in reportgen_txt.seq
XML	AddReportHeader sequence in reportgen_xml.seq	AddReportFooter sequence in reportgen_xml.seq

Table A-4 lists the default process model sequences and C functions in the <TestStand>\Components\Models\TestStandModels directory that generate the report body for each step result.

Table A-4. Sequences or C Functions that Generate the Report Body

Report Format Generator	Description
ATML	GetATMLReport sequence in reportgen_atml.seq calls the Get_Atml_Report function in ATML_Report.c in the ATMLSupport.prj LabWindows/CVI project, located in the <TestStand>\Components\Models\TestStandModels\ATML directory
HTML sequence	AddReportBody sequence in reportgen_html.seq, which indirectly calls the PutOneResultInReport sequence for each result
HTML DLL	PutOneResultInReport_Html function in c_report.c in the modelsupport2.prj LabWindows/CVI project
Text sequence	AddReportBody sequence in reportgen_txt.seq, which indirectly calls the PutOneResultInReport sequence for each result
Text DLL	PutOneResultInReport_Txt function in c_report.c in the modelsupport2.prj LabWindows/CVI project
XML	AddReportBody sequence in reportgen_xml.seq calls the TestStand API PropertyObject.GetXML method

Table A-5 lists the report generation Model callbacks you can override to alter the report generation for each client sequence file you run.

Table A-5. Report Generation Model Callbacks

Section of Report to Alter	Model Callback Sequence to Override
Header	ModifyReportHeader
Footer	ModifyReportFooter
Each step result	ModifyReportEntry TestStand does not call this callback if you select DLL in the Select a Report Generator for Producing the Report Body section of the Contents tab of the Report Options dialog box.
Entire report	TestReport
Batch header	ModifyBatchReportHeader
Batch footer	ModifyBatchReportFooter
Each test socket result	ModifyBatchReportEntry
Entire batch report	BatchReport

Additionally, you can use the Step.Result.ReportText property for each step in a client sequence file to add text to the step result in the report.

Synchronization Step Types

Use synchronization step types to pass data between and perform other operations in multiple threads of an execution, multiple running executions in the same process, and executions running in different processes or on separate computers.

In the TestStand Sequence Editor, use the edit tab on the Step Settings pane to configure Synchronization step types. Select an operation for the step to perform and specify the settings for the operation you select. Some operations store output values to variables you specify. You can leave optional outputs empty. You do not write code modules for Synchronization steps.

In a TestStand User Interface, right-click the step and select **Configure <step type>** from the context menu to configure Synchronization step types. You can also click the **Configure <step type>** button on the General tab of the Step Properties dialog box.

Refer to the *NI TestStand Help* for more information about the edit tabs and configuration dialog boxes for Synchronization step types. Refer to the sequence files in the <TestStand Public>\Examples\Synchronization directory for examples of how to use the Synchronization step types.

Synchronization Objects

Most of the TestStand Synchronization step types create and control the following types of Synchronization objects:

- **Lock**—Use a Lock object to guarantee exclusive access to a resource. For example, if several execution threads write to a device that does not have a thread-safe driver, use a Lock object to ensure that only one thread accesses the device at a time.
- **Rendezvous**—Use a Rendezvous object to make threads wait for each other before proceeding past a location you specify. For example, if different threads configure different aspects of a testing environment, use a Rendezvous object to ensure that none of the threads proceed

beyond the configuration process until all threads complete the configuration tasks.

- **Queue**—Use a Queue object to pass data from the thread that produces the data to a thread that processes the data. For example, a thread that performs tests asynchronously with respect to the Main sequence might use a queue to receive commands from the Main sequence.
- **Notification**—Use a Notification object to notify threads when an event or condition occurs. For example, if you display a dialog box in a separate thread, use a Notification object to notify another thread when the user dismisses the dialog box.
- **Batch**—Use a Batch object to define and synchronize a group of threads, which is useful when you want to test a group of similar UUTs simultaneously. You can configure a synchronized section so that only one UUT enters the section at a time, no UUTs enter the section until all are ready, or no UUTs proceed beyond the section until all UUTs complete, which is useful when, for a particular test, you have only one test resource you must apply separately to each UUT. You can also configure a synchronized section so only one thread executes the steps in the section, which is useful for an action that applies to the entire batch, such as raising the temperature in an environmental chamber. When you control a separate thread for each UUT, you can exploit parallelism and enforce serialization when necessary. You can also use preconditions and other branching options so each UUT can use its own flow of execution.

The TestStand Batch process model creates Batch objects. The model uses Batch Specification steps to group test socket execution threads together so you can use Batch Synchronization steps to synchronize the threads in a sequence file. If you want to create a synchronized section for a single step, use the Synchronization panel on the Properties tab of the Step Settings pane instead of using Batch Synchronization steps.

Refer to the *Batch Process Model* section of Appendix A, *Process Model Architecture*, for more information about the Batch process model. Refer to the *Batch Synchronization* section of this appendix for more information about batch synchronization. Refer to the *NI TestStand Help* for more information about the Synchronization panel on the Properties tab of the Step Settings pane.

- **Semaphore**—Use a Semaphore object to limit access to a resource to a specific number of threads. A Semaphore object is similar to a Lock object except a Semaphore object restricts access to the number of threads you specify rather than to just one thread. For example, use a Semaphore object to restrict access to a communications channel to a

limited number of threads so that each thread has sufficient bandwidth. Typically, you limit access to a shared resource to only one thread at a time. Therefore, typical applications use Lock objects instead of Semaphore objects.

Common Attributes of Synchronization Objects

You can configure each Synchronization step type to specify a name, lifetime settings, and timeout settings for all Synchronization objects.

Name

When you create a Synchronization object, specify a unique name with a literal string or an expression that evaluates to a string to create a reference to the new Synchronization object. Because all named TestStand Synchronization objects share the same name space, you cannot create Synchronization objects with the same name. Synchronization object names are not case sensitive.

If an object with the same name and type already exists, the step creates a reference to the existing Synchronization object so you can access an existing object from multiple threads or executions.

If you specify an empty string as the name for a Synchronization object, TestStand creates an unnamed Synchronization object you can access only through an object reference variable. To associate an unnamed Synchronization object with an object reference variable, select **Use Object Reference** as the object reference lifetime on the edit tab of the Step Settings pane for each step type.

By default, you can access a Synchronization object only from the operating system process in which you create the object. However, you can make a Synchronization object accessible from other processes, such as multiple instances of a user interface, by using an asterisk (*) as the first character in the name.

You can also create a Synchronization object on a specific computer by beginning the name of the object with the computer name, such as \\computername\syncobjectname, which you can use to access the Synchronization object from any computer on the network. To access Synchronization objects on other computers, you must configure DCOM for the TSAutoMgr.exe server, located in the <TestStand>\Bin directory. Follow the instructions in the [Setting up TestStand as a Server for Remote Sequence Execution](#) section of Chapter 5, *Module Adapters*, but apply the instructions to the TSAutoMgr.exe server instead of the

REngine.exe server to configure DCOM and to set up a server for remote access.



Note When you use a string constant in a dialog box expression control to specify an object on a remote computer, escape the backslashes and surround the name in quotation marks. For example, use "\\computername\\syncobjectname".

Lifetime

When you create a Synchronization object, you must specify a lifetime for the reference you create. The object exists for at least as long as the reference exists but can exist longer if another reference to the object specifies a longer lifetime.

You can set the object reference lifetime to Same as Sequence, Same as Thread, Same as Execution, or Use Object Reference. If you refer to the object by name only, you typically set the reference lifetime to Same as Sequence, Same as Thread, or Same as Execution to guarantee that the object lives as long as the sequence, thread, or execution in which you create the reference.

If you want to explicitly control the lifetime of the object reference or if you want to use an object reference variable to refer to the object instead of using the object name, set the lifetime to Use Object Reference. You can also use the object reference from other threads without performing a Create operation in each thread. When the last object reference to a Synchronization object releases, TestStand disposes of the object.

Some Synchronization objects use Lock or Acquire operations, for which you can also specify a lifetime to determine the duration of the operation.

Timeout

Most Synchronization objects can perform operations that time out if the operations do not complete within the number of seconds you specify. You can specify that TestStand treats a timeout as an error condition, or you can explicitly check the value of the Step.Result.TimeoutOccurred property to see if a timeout occurred.

Resource Usage Profiler

Select **Tools»Profile Resource Usage** to launch the Resource Usage Profiler window to view and record the resources a multithreaded TestStand system uses over a period of time. The profiler records resource usage and TestStand thread synchronization operations the system performs as long as the Resource Usage Profiler window is open.

You can review the recorded data in graphs and sortable tables to identify performance bottlenecks and design flaws and to gain insight into the behavior and timing of complex multithreaded systems. You can also copy the information to external applications, such as Microsoft Word or Excel.

Refer to the `Comparing Resource Usage Strategies.seq` file in the `<TestStand Public>\Examples\ResourceUsageProfiler` directory for an example of how to use the profiler. Each example sequence file automatically launches the profiler and displays instructions for the example.

The Resource Usage Profiler displays the use of resources associated with the Auto Schedule, Lock, Rendezvous, Queue, Notification, Wait, Batch Synchronization, and Semaphore Synchronization step types.

Refer to the *NI TestStand Help* for more information about the Resource Usage Profiler.

Lock



Use a Lock step, shown at left, to ensure that only one thread can access a particular resource or data item at a time. For example, if you examine and update the value of a global variable from multiple threads or executions, use a lock to ensure that only one thread examines and updates the variable at a time. Multiple threads waiting to lock an item wait in FIFO (first-in-first-out) order for the item to become available.

A thread can lock the same item an unlimited number of times without unlocking it, but to release the lock, the thread must balance each Lock operation with an Unlock operation.

If all the threads that use a set of locks reside on the same computer and if all the locks in the set reside on that same computer, TestStand detects and reports a run-time error if deadlock occurs with the locks and threads. To avoid deadlock, a set of locks must follow the same order in every thread, or you must use a Lock operation to specify an array of lock names or

references that includes all the locks a thread requires. You can also use the Synchronization panel on the Properties tab of the Step Settings pane to create a lock around a single step.



Note Accessing TestStand variables and properties is thread safe.

Step Properties

In addition to the common custom properties, the Lock step type defines the following step properties:

- **Step.Result.TimeoutOccurred**—Exists only when you configure the step for the Lock operation. TestStand sets the value to `True` if the lock operation times out.
- **Step.NameOrRefExpr**—Contains the Lock Name expression for the Create operation and the Lock Name or Reference expression for all other Lock operations. For the Lock operation, the expression can also specify an array of names or references.
- **Step.LifetimeRefExpr**—Contains the object reference expression for the Lock Reference Lifetime or the Lock Operation Lifetime when you set either lifetime to Use Object Reference.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Lock operation.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Lock operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Lock operation.
- **Step.AlreadyExistsExpr**—Contains the Already Exists expression for the Create operation or the Lock Exists expression for the Get Status operation.
- **Step.NumThreadsWaitingExpr**—Contains the Number of Threads Waiting to Lock the Lock expression for the Get Status operation.
- **Step.Operation**—Contains a value that specifies the operation for the step to perform. The valid values are 0 = Create, 1 = Lock, 2 = Early Unlock, and 3 = Get Status.
- **Step.Lifetime**—Contains a value that specifies the lifetime setting to use for the Create operation. The valid values are 0 = Same as Sequence, 1 = Same as Thread, 2 = Use Object Reference, and 3 = Same as Execution.
- **Step.LockLifetime**—Contains a value that specifies the lifetime setting to use for the Lock operation. The valid values are 0 = Same

as Sequence, 1 = Same as Thread, 2 = Use Object Reference, and 3 = Same as Execution.

- **Step.CreateIfDoesNotExist**—Contains the Create If Does Not Exist setting for the Lock operation.

Rendezvous



Use a Rendezvous step, shown at left, to make threads wait for each other before proceeding past a location you specify. Each thread blocks as it performs the Rendezvous operation. When the number of blocked threads reaches the total number you specified when you created the rendezvous, the rendezvous unblocks all the waiting threads, and the threads resume execution.

Step Properties

In addition to the common custom properties, the Rendezvous step type defines the following step properties:

- **Step.Result.TimeoutOccurred**—Exists only when you configure the step for the Rendezvous operation. TestStand sets the value to `True` if the Rendezvous operation times out.
- **Step.NameOrRefExpr**—Contains the Rendezvous Name expression for the Create operation and the Rendezvous Name or Reference expression for all other Rendezvous operations.
- **Step.LifetimeRefExpr**—Contains the object reference expression for the Rendezvous Reference Lifetime when you set the lifetime to Use Object Reference.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Rendezvous operation.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Rendezvous operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Rendezvous operation.
- **Step.AlreadyExistsExpr**—Contains the Already Exists expression for the Create operation or the Rendezvous Exists expression for the Get Status operation.
- **Step.RendezvousCountExpr**—Contains the Number of Threads Per Rendezvous expression for the Create operation.
- **Step.NumThreadsWaitingExpr**—Contains the Number of Threads Waiting for Rendezvous expression for the Get Status operation.

- **Step.Operation**—Contains a value that specifies the operation for the step to perform. The valid values are 0 = Create, 1 = Rendezvous, and 2 = Get Status.
- **Step.Lifetime**—Contains a value that specifies the lifetime setting to use for the Create operation. The valid values are 0 = Same as Sequence, 1 = Same as Thread, 2 = Use Object Reference, and 3 = Same as Execution.
- **Step.RendezvousCountOutExpr**—Contains the Number of Threads Per Rendezvous expression for the Get Status operation.

Queue



Use Queue steps, shown at left, to synchronize the production and consumption of data among threads. An Enqueue operation places a data item on the queue and blocks when the queue is full. A Dequeue operation removes an item from the queue and blocks when the queue is empty. If multiple threads block on the same Queue operation, the threads unblock in FIFO order.

Step Properties

In addition to the common custom properties, the Queue step type defines the following step properties:

- **Step.Result.TimeoutOccurred**—Exists only when you configure the step for the Enqueue or Dequeue operation. TestStand sets the value to `True` if an Enqueue or Dequeue operation times out.
- **Step.NameOrRefExpr**—Contains the Queue Name expression for the Create operation and the Queue Name or Reference expression for all other Queue operations. For the Dequeue operation, the expression can also specify an array of names or references.
- **Step.LifetimeRefExpr**—Contains the object reference expression for the Queue Reference Lifetime when you set the lifetime to Use Object Reference.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Enqueue or Dequeue operation.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Enqueue or Dequeue operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Enqueue or Dequeue operation.

- **Step.AlreadyExistsExpr**—Contains the Already Exists expression for the Create operation or the Queue Exists expression for the Get Status operation.
- **Step.MaxNumElementsExpr**—Contains the expression that specifies the maximum number of queue elements for the Create operation.
- **Step.MaxNumElementsOutExpr**—Contains the expression that specifies where to store the maximum number of queue elements for the Get Status operation.
- **Step.NumThreadsWaitingEnqueueExpr**—Contains the expression that specifies where to store the number of threads waiting to enqueue for the Get Status operation.
- **Step.NumThreadsWaitingDequeueExpr**—Contains the expression that specifies where to store the number of threads waiting to dequeue for the Get Status operation.
- **Step.Operation**—Contains a value that specifies the operation for the step to perform. The valid values are 0 = Create, 1 = Enqueue, 2 = Dequeue, 3 = Flush, and 4 = Get Status.
- **Step.Lifetime**—Contains a value that specifies the lifetime setting to use for the Create operation. The valid values are 0 = Same as Sequence, 1 = Same as Thread, 2 = Use Object Reference, and 3 = Same as Execution.
- **Step.NumElementsExpr**—Contains the expression that specifies where to store the current number of queue elements for the Get Status operation.
- **Step.DataExpr**—Contains the New Element to Enqueue expression for the Enqueue operation, the Location to Store Element expression for the Dequeue operation, and the Location to Store Array of Queue Elements expression for the Flush or Get Status operation.
- **Step.ByRef**—Contains the Boolean value that specifies to store a queue element by object reference instead of by value for the Enqueue operation.
- **Step.EnqueueLocation**—Contains a value that specifies the location to store the queue element for the Enqueue operation. The valid values are 0 = Front of Queue and 1 = Back of Queue.
- **Step.DequeueLocation**—Contains a value that specifies the location from which to remove the queue element for the Dequeue operation. The valid values are 0 = Front of Queue and 1 = Back of Queue.

- **Step.FullQueueOption**—Contains a value that specifies the options for the If the Queue is Full setting of the Enqueue operation. The valid values are 0 = Wait, 1 = Discard Front Element, 2 = Discard Back Element, and 3 = Do Not Enqueue.
- **Step.RemoveElement**—Contains a Boolean value that specifies to remove the element from the queue when the step performs the Dequeue operation.
- **Step.WhichQueueExpr**—Contains the expression that specifies where to store the array offset of the queue on which the Dequeue operation occurs.

Notification



Use Notification steps, shown at left, to notify threads when a particular event or condition occurs. You can also pass data to the threads you notify.

Step Properties

In addition to the common custom properties, the Notification step type defines the following step properties:

- **Step.Result.TimeoutOccurred**—Exists only when you configure the step for the Wait operation. TestStand sets the value to `True` if a Wait operation times out.
- **Step.NameOrRefExpr**—Contains the Notification Name expression for the Create operation and the Notification Name or Reference expression for all other Notification operations. For the Wait operation, the expression can also specify an array of names or references.
- **Step.LifetimeRefExpr**—Contains the object reference expression for the Notification Reference Lifetime when you set the lifetime to Use Object Reference.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Wait operation.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Wait operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Wait operation.
- **Step.AlreadyExistsExpr**—Contains the Already Exists expression for the Create operation or the Notification Exists expression for the Get Status operation.

- **Step.NumThreadsWaitingExpr**—Contains the expression that specifies where to store the number of threads waiting on the notification for the Get Status operation.
- **Step.Operation**—Contains a value that specifies the operation for the step to perform. The valid values are 0 = Create, 1 = Set, 2 = Clear, 3 = Pulse, 4 = Wait, and 5 = Get Status.
- **Step.Lifetime**—Contains a value that specifies the lifetime setting to use for the Create operation. The valid values are 0 = Same as Sequence, 1 = Same as Thread, 2 = Use Object Reference, and 3 = Same as Execution.
- **Step.DataExpr**—Contains the Data Value expression for the Set or Pulse operation or the Location to Store Data expression for the Wait or Get Status operation.
- **Step.ByRef**—Contains the Boolean value that specifies to store the data by object reference instead of by value for the Set or Pulse operation.
- **Step.WhichNotificationExpr**—Contains the expression that specifies where to store the array offset of the notification to which the Wait operation responds.
- **Step.IsSetExpr**—Contains the expression that specifies for the Get Status operation where the step stores the Boolean value that indicates the Set state of the notification.
- **Step.IsAutoClearExpr**—Contains the expression that specifies for the Get Status operation where to store the Boolean value that indicates the AutoClear state of the notification.
- **Step.AutoClear**—Contains the AutoClear setting for the Set operation.
- **Step.PulseNotifyOpt**—Contains the setting for the Pulse operation that indicates the threads to send a pulse notification to. The valid values are 0 = Notify First Waiting Thread and 1 = Notify All Waiting Threads.

Wait



Use Wait steps, shown at left, to wait for an execution or thread to complete or to wait for a time interval to elapse.

When the thread or execution completes, the Wait step copies the result status and error information for the thread or execution to its own status and error properties. Therefore, if a Wait step waits on a sequence that fails, TestStand sets the status of the Wait step to `Failed`. The result list entry

for a Wait step contains a `TS.SequenceCall.ResultList` property, which is the result list for the thread or execution.

In a Wait step, do not specify to wait on a Sequence Call step if the Sequence Call step launches more than one asynchronous call, such as in a loop, because the Wait step waits on only the last asynchronous call the Sequence Call step launches. To wait on multiple asynchronous calls you launch from a Sequence Call step in a loop, store an ActiveX reference to each thread or execution you launch and wait on each reference in a Wait step.

Step Properties

In addition to the common custom properties, the Wait step type defines the following step properties:

- **Step.Result.TimeoutOccurred**—Exists only when you configure the step for the Wait for Thread or Wait for Execution operation. TestStand sets the value to `True` if the Wait for Thread or Wait for Execution operation times out.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Wait for Thread or the Wait for Execution operation.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Wait for Thread or the Wait for Execution operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Wait for Thread or the Wait for Execution operation.
- **Step.ThreadRefExpr**—Contains the Thread Reference expression for the Wait for Thread operation when the `Step.SpecifyBySeqCall` property is `False`.
- **Step.SeqCallName**—Contains the name of the Sequence Call step that creates the thread or execution the step waits for when the `Step.SpecifyBySeqCall` property is `True`.
- **Step.SeqCallStepGroupIdx**—Contains the step group of the Sequence Call step that creates the thread or execution the step waits for when the `Step.SpecifyBySeqCall` property is `True`. The valid values are 0 = Setup, 1 = Main, and 2 = Cleanup.
- **Step.WaitForTarget**—Contains a value that specifies the type of Wait operation the step performs. The valid values are 0 = Time Interval, 1 = Time Multiple, 2 = Thread, and 3 = Execution.
- **Step.TimeExpr**—Contains the time expression for the Time Interval or Time Multiple operation of the step.

- **Step.ExecutionRefExpr**—Contains the expression that specifies a reference to the execution on which the Wait for Execution operation waits.
- **Step.SpecifyBySeqCall**—Contains the Specify By Sequence Call setting for the Wait for Thread or the Wait for Execution operation.

At run time, TestStand adds the following properties to the results for Wait steps you configure to wait for a thread or execution:

- **AsyncMode**—TestStand sets the value to `True` if the Wait step is waiting on a thread and to `False` if the Wait step is waiting on an execution.
- **AsyncId**—Contains the value of the `Id` property of the thread or execution the step is waiting for.

Batch Synchronization



Use Batch Synchronization steps, shown at left, to define sections of a sequence in which to synchronize multiple threads that belong to one batch. Place Batch Synchronization steps around test steps to create a synchronized section. Use the Synchronization panel on the Properties tab of the Step Settings pane to synchronize a single step for the multiple threads that belong to a batch. Typically, you use Batch Synchronization steps in a sequence you execute using the Batch process model.

Synchronized Sections

Place a Batch Synchronization step at the beginning of a section of steps in a sequence and specify an Enter operation for the step. Place another Batch Synchronization step at the end of the section of steps and specify an Exit operation for the step. You must place the Enter and Exit steps in the same sequence file, but you do not have to place the Enter and Exit steps in the same step group.

Each thread in a batch that enters a synchronized section blocks at the Enter step until all the other threads in the batch arrive at their respective instances of the Enter step. A thread cannot re-enter a section it has already entered. Each thread in a batch that reaches the end of the synchronized section blocks at the Exit step until all the other threads in the batch arrive at their respective instances of the Exit step.

You can use the following types of synchronized sections in sequence files:

- **Serial**—Use a Serial section to ensure that each thread in the batch executes the steps in the section sequentially and in the order you

specify when you create the batch. When all threads in a batch arrive at their respective instances of an Enter step for a Serial section, TestStand releases one thread at a time in ascending order according to the order numbers you assign to the threads when you use a Batch Specification step to add the threads to the batch. As each thread reaches the Exit step for the section, the next thread in the batch proceeds from the Enter step. After all the threads in the batch arrive at the Exit step, the threads exit the section together. Refer to the [Semaphore](#) section of this appendix for more information about order numbers.

- **Parallel**—Use a Parallel section to run each thread independently. When all threads in a batch arrive at their respective instances of an Enter step for a Parallel section, TestStand releases all the threads at once. As each thread reaches the Exit step for the section, the thread blocks until all the threads in the batch reach the Exit step. After all the threads in the batch arrive at the Exit step, the threads exit the section together.
- **One Thread Only**—Use a One Thread Only section to specify that only one thread in the batch executes the steps in the section. Typically, you use this type of section to perform an operation that applies to the batch as a whole, such as raising the temperature in a test chamber. When all threads in a batch arrive at their respective instances of an Enter step for a One Thread Only section, TestStand releases only one thread. When that thread arrives at the Exit step for the section, all remaining threads in the batch jump from the Enter step to the Exit step, skipping the steps within the section. All the threads in the batch exit the section together.

Mismatched Sections

Sections become mismatched when all the threads in a batch block at different Enter or Exit operations. This situation can occur when a sequence implements a conditional flow of execution as a result of preconditions, post actions, or other flow control operations. When TestStand detects mismatched sections, the thread at the Enter or Exit step that appears earliest in the hierarchy of sequences and subsequences proceeds as if all threads in the batch are at the same step. If multiple Enter and Exit operations are equally early in the hierarchy of sequences and subsequences, Enter operations proceed first.

Nested Sections

Nested sections can occur within the same sequence or when you call a subsequence inside a synchronized section and the subsequence also

contains a synchronized section. You must exit nested sections in the reverse order in which you entered the sections. When you nest one section inside another, TestStand honors the inner section only when the type of the outer section is serial or parallel. TestStand ignores the inner section if the type of the outer section is One Thread Only. For example, if you nest one serial section in another serial section, each thread that enters the outer serial section proceeds only until the Enter step of the inner section and then waits for the other threads to reach the same Enter step of the inner section before completing the inner section.

Step Properties

In addition to the common custom properties, the Batch Synchronization step type defines the following step properties:

- **Step.Result.TimeoutOccurred**—Exists only when you configure the step for the Enter or Exit operation. TestStand sets the value to `True` if an Enter or Exit operation times out.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Enter or Exit operation.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Enter or Exit operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Enter or Exit operation.
- **Step.Operation**—Contains a value that specifies the operation for the step to perform. The valid values are 0 = Enter Synchronized Section and 1 = Exit Synchronized Section.
- **Step.SectionNameExpr**—Contains the expression that specifies the section name for the Enter or Exit operation.
- **Step.SectionType**—Contains a value that specifies the type of section the Enter operation defines. The valid values are 1 = Serial, 2 = Parallel, and 3 = One Thread Only.

Auto Schedule



Use the Auto Schedule step, shown at left, to define a block that contains any number of Use Auto Scheduled Resource step sub-blocks. You typically use Auto Schedule steps in a sequence you execute using the Parallel or Batch process models. The Auto Schedule step executes each sub-block once. The order in which the Auto Schedule step executes the sub-blocks can vary according to the availability of the resources the sub-blocks require. The Auto Schedule step can increase CPU and resource

use by directing a thread that otherwise waits for a resource another thread locks to perform other actions using available resources instead.

Refer to the `<TestStand Public>\Examples\Auto Schedule` directory for examples of how to use Auto Schedule and Use Auto Scheduled Resource steps.

Step Properties

In addition to the common custom properties, the Auto Schedule step type defines the following step properties:

- **Step.Result.TimeoutOccurred**—Exists only when you configure the step for the Acquire operation. TestStand sets the value to `True` if any Auto Scheduled Resource blocks within the Auto Schedule block time out.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Auto Schedule operation.
- **Step.TimeoutExpression**—Contains the Timeout expression, in seconds, for the Auto Schedule operation.
- **Step.TimeoutIsRuntimeError**—Setting value to `True` causes a step run-time error when a timeout occurs.
- **Step.DisplayRuntimeDescription**—Setting to `True` displays execution scheduling information in the step description.

Use Auto Scheduled Resource



Use the Use Auto Scheduled Resource step, shown at left, to define a sub-block of steps within an Auto Schedule block that uses a resource or set of resources you specify. The Use Auto Scheduled Resource step locks the resources you specify while the steps in its sub-block execute.

Refer to the `<TestStand Public>\Examples\Auto Schedule` directory for examples of how to use Auto Schedule and Use Auto Scheduled Resource steps.

Step Properties

In addition to the common custom properties, the Use Auto Scheduled Resource step type defines the following step properties:

- **Step.ResourceExpressions**—Contains a list of expressions that specify the lock alternatives the block can acquire before executing the steps within the block.

- **Step.SelectedResourceExpression**—Contains an expression that specifies a string variable or property into which to store the lock the step acquires during execution.

Thread Priority



Use the Thread Priority step, shown at left, to adjust the operating system priority of a TestStand thread so that the TestStand thread receives more or less CPU time than other threads. Avoid starving important threads of CPU time by boosting the priority of another thread too high. For example, setting the priority of a thread to Time Critical can cause the user interface of an application to become unresponsive. When you alter a thread priority, save the previous priority value and restore it when the changed thread no longer requires the altered priority value.

Step Properties

In addition to the common custom properties, the Thread Priority step type defines the following step properties:

- **Step.Operation**—Contains a value that specifies the operation for the step to perform. The valid values are 0 = Set Thread Priority and 1 = Get Thread Priority.
- **Step.SetPriorityExpr**—Specifies the thread priority expression for the Set Thread Priority operation.
- **Step.GetPriorityExpr**—Specifies the location to store the thread priority for the Get Thread Priority operation.

Semaphore



Use Semaphore steps, shown at left, to limit concurrent access to a resource to a specific number of threads. A semaphore stores a numeric count, and threads can increment (release) or decrement (acquire) the count as long as the count stays equal to or greater than zero. If a decrement causes the count to drop below zero, the thread that attempts to decrement the count blocks until the count increases. When multiple threads wait to decrement a semaphore and another thread increments the count, the semaphore unblocks the threads in FIFO order.

A semaphore with an initial count of one behaves like a lock because a one-count semaphore restricts access to a single thread at a time. Unlike a lock, however, a thread cannot acquire a one-count semaphore multiple times without first releasing the semaphore after each acquire. When a

thread attempts to acquire the semaphore a second time without releasing it, the count is zero, and the thread blocks. Refer to the [Lock](#) section of this appendix for more information about Lock objects.

Step Properties

In addition to the common custom properties, the Semaphore step type defines the following step properties:

- **Step.Result.TimeoutOccurred**—Exists only when you configure the step for the Acquire operation. TestStand sets the value to `True` if the Acquire operation times out.
- **Step.NameOrRefExpr**—Contains the Semaphore Name expression for the Create operation and the Semaphore Name or Reference expression for all other Semaphore operations.
- **Step.AutoRelease**—Contains a Boolean value that specifies if the Acquire operation automatically performs a release when the Acquire lifetime expires.
- **Step.LifetimeRefExpr**—Contains the object reference expression for the Semaphore Reference Lifetime or the Acquire Reference Lifetime when you set either lifetime to Use Object Reference.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Acquire operation.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Acquire operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Acquire operation.
- **Step.AlreadyExistsExpr**—Contains the Already Exists expression for the Create operation or the Semaphore Exists expression for the Get Status operation.
- **Step.InitialCountExpr**—Contains the Initial Semaphore Count expression for the Create operation.
- **Step.NumThreadsWaitingExpr**—Contains the Number of Threads Waiting to Acquire the Semaphore expression for the Get Status operation.
- **Step.Operation**—Contains a value that specifies the operation for the step to perform. The valid values are 0 = Set Thread Priority and 1 = Get Thread Priority.

- **Step.Lifetime**—Contains a value that specifies the lifetime setting to use for the Create operation. The valid values are 0 = Same as Sequence, 1 = Same as Thread, 2 = Use Object Reference, and 3 = Same as Execution.
- **Step.InitialCountOutExpr**—Contains the Initial Semaphore Count expression for the Get Status operation.
- **Step.AcquireLifetime**—Contains a value that specifies the lifetime setting for the Acquire operation. The valid values are 0 = Same as Sequence, 1 = Same as Thread, and 2 = Use Object Reference. The Acquire operation uses this setting only when Step.AutoRelease is True.
- **Step.CurrentCountExpr**—Contains the Current Count expression for the Get Status operation.

Batch Specification



Use Batch Specification steps, shown at left, to define a group of threads in which each thread in the group runs an instance of the client sequence file. When you define a group, you can perform Batch Synchronization operations on the threads in the group. The TestStand Batch process model uses Batch Specification steps to create a batch that contains a thread for each test socket.

When you test each UUT in a separate thread, use the Batch Specification step to include the UUT threads in one batch. Use the Batch Synchronization step to control the interaction of the UUT threads as they execute the test steps.

Refer to the [Batch Synchronization](#) section of this appendix for more information about batch synchronization. Refer to the [Batch Process Model](#) section of Appendix A, [Process Model Architecture](#), for more information about the Batch process model.

Step Properties

In addition to the common custom properties, the Batch Specification step type defines the following step properties:

- **Step.Operation**—Contains a value that specifies the operation for the step to perform. The valid values are 0 = Create, 1 = Add Thread, 2 = Remove Thread, and 3 = Get Status.

- **Step.NameOrRefExpr**—Contains the Name expression for the Create operation and the Name or Reference expression for all other Batch operations.
- **Step.Lifetime**—Contains a value that specifies the lifetime setting to use for the Create operation. The valid values are 0 = Same as Sequence, 1 = Same as Thread, 2 = Use Object Reference, and 3 = Same as Execution.
- **Step.LifetimeRefExpr**—Contains the object reference expression for the Batch Reference Lifetime when you set the lifetime to Use Object Reference.
- **Step.AlreadyExistsExpr**—Contains the Already Exists expression for the Create operation or the Batch Exists expression for the Get Status operation.
- **Step.ThreadRefExpr**—Contains the Object Reference to Thread expression for the Add Thread and Remove Thread operations.
- **Step.OrderNumExpr**—Contains the Order Number expression for the Add Thread operation.
- **Step.NumThreadsWaitingExpr**—Contains the Number of Threads Waiting at Synchronized Sections expression for the Get Status operation.
- **Step.NumThreadsInBatchExpr**—Contains the Number of Threads in Batch expression for the Get Status operation.
- **Step.DefaultBatchSyncExpr**—Contains the Default Batch Synchronization expression for the Create operation.
- **Step.DefaultBatchSyncOutExpr**—Contains the Default Batch Synchronization expression for the Get Status operation.



Database Step Types



Use the built-in Database step types, to communicate with a database. All the Database step types, with the exception of the Property Loader step type, use the icon shown at left. A simple database operation includes the following steps:

1. Use the Open Database step type to connect to a database.
2. Use the Open SQL Statement step type to perform an SQL query on tables in the database.
3. Use Data Operation step types to create new records and to retrieve and update existing records.
4. Use the Close SQL Statement step type to close the SQL query.
5. Use the Close Database step type to disconnect from a database.

Use the Property Loader step type to import property and variable values from a file or database during an execution.

Right-click the Database step and select **Edit <step type>** from the context menu to configure the step type and to set the custom step properties. You can also click the **Edit <step type>** button on the edit tab of the Step Settings pane.

Refer to the <TestStand Public>\Examples\Database directory and to the <TestStand Public>\Examples\Property Loader directory for examples of how to use the Database step types. Refer to the *NI TestStand Help* for more information about the Database step types.

Open Database

Use the Open Database step type to open a database for use in TestStand. The Open Database step returns a database handle you can use to open SQL statements.

Step Properties

In addition to the common custom properties, the Open Database step type defines the following step properties:

- **Step.ConnectionString**—Specifies a string expression that contains the name of the data link to open.
- **Step.DatabaseHandle**—Specifies the numeric variable or property the step type assigned as the value of the opened database handle.

Open SQL Statement

After you open a database, use the Open SQL Statement step type to select the set of data with which to work. An Open SQL Statement step returns a statement handle you can use in Data Operation steps.

Step Properties

In addition to the common custom properties, the Open SQL Statement step type defines the following step properties:

- **Step.PageSize**—Specifies the number of records in a page for the SQL statement.
- **Step.CommandTimeout**—Specifies the amount of time, in seconds, TestStand waits while attempting to issue a command to the open database connection.
- **Step.CacheSize**—Specifies the cache size for the SQL statement.
- **Step.MaxRecordsToSelect**—Specifies the maximum number of records the SQL statement can return.
- **Step.CursorType**—Specifies the cursor type the SQL statement uses.
- **Step.CursorLocation**—Specifies where the data source maintains cursors for a connection.
- **Step.MarshalOptions**—Specifies the marshal options for the updated records in the SQL statement.
- **Step.LockType**—Specifies the lock type for the records the SQL statement selects.
- **Step.CommandType**—Specifies the command type of the SQL statement.
- **Step.DatabaseHandle**—Specifies the name of the variable or property that contains the database handle with which you open the SQL statement.

- **Step.StatementHandle**—Specifies the numeric variable or property the step type assigned as the value of the SQL statement handle.
- **Step.SQLStatement**—Specifies a string expression that contains the SQL command.
- **Step.NumberOfRecordsSelected**—Specifies the numeric variable or property to which the step assigns the number of records the SQL statement returns.
- **Step.RequiresParameters**—Specifies if the SQL statement requires input or output parameters to execute. If `False`, the step immediately executes the SQL statement. If `True`, the step only prepares the SQL statement, and a subsequent Data Operation step must perform an Execute operation that defines the parameters for the statement.

Close SQL Statement

Use the Close SQL Statement step to close an SQL statement handle you obtain from an Open SQL Statement step. National Instruments recommends placing Close SQL Statement steps in the Cleanup step group. Refer to the [Step Groups](#) section of Chapter 1, *NI TestStand Architecture*, for more information about step groups.

Step Properties

In addition to the common custom properties, the Close SQL Statement step type defines the `Step.StatementHandle` step property, which specifies the name of the numeric variable or property that contains the SQL statement handle to close.

Close Database

Use the Close Database step type to close the database handle you obtain from an Open Database step type. You must call a Close Database step for open handles because TestStand does not automatically close open database handles. If you abort an execution, you must exit the application process that loaded the TestStand Engine to guarantee that TestStand frees all database handles. Selecting Unload All Modules from the File menu does not close the handles. National Instruments recommends placing Close Database steps in the Cleanup step group. Refer to the [Step Groups](#) section of Chapter 1, *NI TestStand Architecture*, for more information about step groups.

Step Properties

In addition to the common custom properties, the Close Database step type defines the `Step.DatabaseHandle` step property, which specifies the name of the numeric variable or property that contains the open database handle to close.

Data Operation

Use the Data Operation step type to perform operations on an SQL statement you open with an Open SQL Statement step. Use the Data Operation step to fetch new records, retrieve values from a record, modify existing records, create new records, and delete records. For SQL statements that require parameters, you can create parameters and set input values, execute statements, close statements, and fetch output parameter values.

You cannot encapsulate data operations within a transaction because the current TestStand Database step types do not support transactions.

Step Properties

In addition to the common custom properties, the Data Operation step type defines the following step properties:

- **Step.StatementHandle**—Specifies a string expression that contains the name of the SQL statement on which to operate.
- **Step.RecordToOperateOn**—Specifies the record on which to operate. Valid values are 0 = New, 1 = Current, 2 = Next, 3 = Previous, and 4 = Index.
- **Step.RecordIndex**—Specifies the index of the record on which to operate when you set the `Step.RecordToOperateOn` property to fetch a specific index.
- **Step.Operation**—Specifies the operation to perform on the record. Valid values are 0 = Fetch only, 1 = Set, 2 = Get, 3 = Put, 4 = Delete, 5 = Set and Put, 6 = Execute, and 7 = Close.
- **Step.ColumnListSource**—Specifies the name of the `DatabaseColumnValue` array variable or property that stores the column-to-variable or column-to-property mappings. By default, the value is `Step.ColumnList`.

- **Step.ColumnList**—Specifies the column-to-variable or column-to-property mapping to perform on a Get or Set operation. The property must be an array of DatabaseColumnValue custom data types, which contain the following subproperties:
 - **ColumnName**—Specifies the name or number of the column from which to get a value or to assign a value to.
 - **Data**—Specifies the variable or property to which TestStand assigns the column value or the expression TestStand evaluates and assigns to the column.
 - **FormatString**—Specifies an optional format string for dates, times, and currencies. Use the empty string (" ") to use the default format. Refer to the *NI TestStand Help* for a description of valid format strings.
 - **WriteNull**—Specifies if TestStand writes NULL to the column instead of the value in the Data expression property.
 - **Status**—Indicates the error code TestStand returns for the Get or Set operation.
 - **Direction**—Contains an enumerated value that specifies the parameter direction as In, Out, In/Out, or Return.
 - **Type**—Contains an enumerated value that specifies the parameter value as String, Number, Boolean, or Date/Time.
 - **Size**—Specifies the maximum size of a string parameter.
- **Step.SQLStatement**—Specifies the SQL statement the Edit Data Operation dialog box uses to populate the ring controls that contain column names.

Property Loader



Use the Property Loader step type, shown at left, to dynamically load property and variable values from a text file, a Microsoft Excel file, or a DBMS database at run time. The Property Loader step type can load limits from all TestStand-supported databases except MySQL.

You can apply the values you load to the current sequence. For example, you can develop a sequence that tests two different models of a cellular phone, where each model requires unique limit values for each step. If you use step properties to hold the limit values, include a Property Loader step in the Setup step group of the sequence to initialize the property and variable values each time before the steps in the Main step group execute.

You can also load values for properties into sequences so that all subsequent invocations of the sequences in the file use the dynamically loaded property values. For example, include the Property Loader step in a ProcessSetup model callback the execution calls once so the execution can call the client sequence file multiple times with the dynamically loaded property values.

Loading from Files

You can use tab-delimited text files (.txt), comma-delimited text files (.csv), or Excel files (.xls) to load limit values. The following tab-delimited limits text file includes one data block starting and ending data markers specify.

```
Start Marker

<Step Name>           Limits.Low      Limits.High      Limits.String
Voltage at Pin A      9.0           11.0
Voltage at Pin B      8.5           9.5
Self Test Output                                "SYS OK"

<Locals>
Count                Variable Value
                   100

<FileGlobals>
Count                Variable Value
                   99

<StationGlobals>
Power_On             Variable Value
                   False

End Marker
```

In the step name section of this example file, the row names correspond to step names, and the column headings correspond to the names of step properties. Each row contains values only for the columns that define properties that exist in the step that corresponds to that row.

In the locals, file globals, and station globals variable sections, each row specifies the name of the property and corresponding property value.

Starting and ending data markers designate the bounds of the block of data. A data file can contain more than one block of data.

Select **Tools»Import/Export Properties** to export property and variable data in the appropriate block format. When you specify starting and ending data markers in the text controls in the Import/Export Properties dialog box, enter the marker text without double quotation marks. When you specify starting and ending data markers in the expression controls in the Edit Property Loader dialog box, you must surround literal marker text values with double quotation marks. Refer to the examples in the `<TestStand Public>\Examples\Property Loader\LoadingLimits` directory and the *NI TestStand Help* for more information about loading limits from files.

Loading from Databases

You can use the recordset an Open SQL Statement step returns to load limit values. Each row of the recordset table pertains to a particular sequence step or to a variable scope, as shown in Table C-1. The column headings correspond to the names of properties in the steps or variable scopes. Each row contains values only for the columns that define properties or variables that exist in the step or variable scope that corresponds to that row.

Table C-1. Example Data for Property Loader Step

STEPNAME	LIMITS_ HIGH	LIMITS_ LOW	LIMITS_ STRING	POWER_ON	COUNT	SEQUENCE NAME
Voltage at Pin A	9.0	11.0	—	—	—	Phone Test.seq
Voltage at Pin B	8.5	9.5	—	—	—	Phone Test.seq
Self Test Output	—	—	"SYS OK"	—	—	Phone Test.seq
<Locals>	—	—	—	—	100	Phone Test.seq
<File Globals>	—	—	—	—	99	Phone Test.seq
<Station Globals>	—	—	—	False	—	Phone Test.seq
Frequency at Pin A	100,000	10,000	—	—	—	Frequency Test.seq
Frequency at Pin B	90,000	9,000	—	—	—	Frequency Test.seq
Self Test Output	—	—	"OK"	—	—	Frequency Test.seq

The Property Loader step filters the data an SQL statement returns so you load only values from rows that contain specific column values, which is equivalent to using starting and ending data markers in a text or Excel file. For example, you can load only the rows in Table C-1 where the SEQUENCE NAME field contains the value `Phone Test.seq`.

Refer to the example in the <TestStand Public>\Examples\Property Loader directory and the *NI TestStand Help* for more information about loading limits from database tables.

Step Properties

In addition to the common custom properties, the Property Loader step type defines the following step properties:

- **Step.Result.NumPropertiesRead**—Indicates the total number of values the step loaded from the file or database.
- **Step.Result.NumPropertiesApplied**—Indicates the total number of values the step assigned to properties or variables. A number less than Step.Result.NumPropertiesRead indicates the step was unable to update properties or variables.
- **Step.ColumnListSource**—Specifies the name of the DatabaseColumnValue array variable or property that stores the list of column comparisons you use to filter the rows in a database recordset. By default, the value is Step.ColumnList.
- **Step.ColumnList**—Specifies the column comparisons TestStand makes on a recordset before TestStand loads recordset values into a property. The property must be an array of DatabaseColumnValue custom data types, which contain the following subproperties:
 - **ColumnName**—Specifies the name or number of the column on which to perform the comparison.
 - **Data**—Specifies the expression TestStand evaluates at run time to compare against the column value.
 - **FormatString**—Specifies an optional format string for dates, times, and currencies. Use an empty string (" ") to use the default format. Refer to the *NI TestStand Help* for a description of valid format strings.
 - **Direction**—Contains an enumerated value that specifies the parameter direction as In, Out, In/Out, or Return.
 - **Type**—Contains an enumerated value that specifies the parameter type as String, Number, Boolean, or Date/Time.
 - **Size**—Specifies the maximum size of a string parameter.
 - **WriteNull**—Not used.
 - **Status**—Not used.

- **Step.PropertiesListSource**—Specifies the name of the DatabasePropertyMapping array variable or property that stores the list of variables and properties in which to load data. By default, the value is Step.PropertiesList.
- **Step.PropertiesList**—Specifies the list of variables and properties in which to load data. The list must be an array of DatabasePropertyMapping custom data types. Each element of the array defines a mapping of source data to a TestStand variable or property. The DatabasePropertyMapping custom data type contains the following subproperties:
 - **PropertyName**—Specifies the name of the property or variable to which TestStand assigns a value.
 - **PropertyType**—Specifies the scope of the property or variable. Valid values are 0 = Step, 1 = Local, 2 = File Global, and 3 = Station Global.
 - **DataType**—Specifies the TestStand type of the property. Valid values are 1 = String, 2 = Boolean, and 3 = Number.
 - **ColumnName**—Specifies the name of the column from which TestStand obtains the value.
- **Step.DataSourceType**—Specifies where the step imports property values from. Valid values are 2 = File and 3 = Database.
- **Step.Database**—Specifies the SQL statement handle and settings for importing property values from a database to a sequence file. The Database step property contains the following subproperties:
 - **SQLStatementHandle**—Specifies the name of the variable or property that contains the SQL statement handle the step uses at run time to load values.
 - **SQLStatement**—Specifies the SQL statement the Edit Property Loader dialog box uses to populate ring controls that contain column names.
 - **StepNameColumn**—Specifies the name of the column in the recordset that contains the names of the steps and variable scopes that define the rows of data.
 - **AppendTypeName**—Specifies if TestStand appends the data type name of the property to the column name when selecting a property from the available list.

- **FilterUsingColumnList**—Specifies if the step loads only the rows that match the specific column value.
- **MaxColumnSize**—Specifies the maximum number of characters for a column name.
- **Step.File**—Specifies the file and settings for importing property values from a file to a sequence file. The File step property contains the following subproperties:
 - **Path**—Specifies a literal pathname for the data file.
 - **DecimalPoint**—Specifies the type of decimal point the file uses.
 - **UseExpr**—Specifies if TestStand uses Step.File.Path or Step.File.FileExpr for the pathname of the data file.
 - **FileExpr**—Specifies a pathname expression TestStand evaluates at run time.
 - **Format**—Specifies the type of delimiters in the file and the file type. Valid values are `Tab`, `Comma`, or `Excel`.
 - **Start.MarkerExpr**—Specifies the expression for the starting marker.
 - **EndMarkerExpr**—Specifies the expression for the ending marker.
 - **Skip**—Specifies the string that causes the step type to ignore the row when the string appears at the beginning of the row.
 - **MapColumnsUsingFirstRow**—Specifies if the first row of each data block in the data file contains the names of the step properties into which the step loads the property values.
 - **ColumnMapping**—Specifies the names of the properties into which the step loads the values if Step.File.MapColumnsUsingFirstRow is `False`.
- **Step.SequenceFile**—Specifies the path to the sequence file to import properties to.
- **Step.Sequence**—Specifies the sequence to which the step imports properties.
- **Step.ExpandToRelatedExecutions**—Specifies that TestStand applies imported property values to sequences running in related executions, which include the original execution and all executions TestStand invoked or invokes using a Sequence Call step.

- **Step.UseCurrentSequence**—Specifies to import properties to the run-time copy of the sequence that includes the step. Otherwise, imported properties apply to all invocations of the sequences the step imports to.
- **Step.UseCurrentFile**— Specifies to import properties to the sequence file that includes the step.
- **Step.ImportAll**—Specifies if the step attempts to import all property values listed in a file into the selected sequence files.
- **Step.StartMarkerMissingAction**—Specifies the action the step takes when TestStand does not find the start marker in the file. Valid values are 1 = Stop and error and 2 = Skip sequence.

IVI Step Types

Interchangeable Virtual Instrument (IVI) is an instrument driver standard that provides common programming interfaces for several classes of instruments. IVI drivers exist for a number of popular instruments, including National Instruments devices. Refer to the National Instruments Web site at ni.com/ivi for more information about IVI. Refer to the Instrument Driver Network at ni.com/idnet for more information about instrument drivers and for finding and downloading instrument drivers compatible with National Instruments software.

Two architectures exist for IVI drivers—IVI-C, based on ANSI C, and IVI-COM, based on Microsoft COM technology. The IVI step types support IVI-C class-compliant instrument drivers, and support IVI-COM class-compliant instrument drivers if you install the IVI-COM Adapter component of the IVI Component Package included in the NI Driver CD. TestStand does not install IVI class instrument drivers.

You can call IVI-C instrument class drivers and specific drivers from any development environment that supports calls into DLLs. Many IVI-C instrument drivers have native LabVIEW-generated wrappers. You can also convert an IVI-C instrument driver using the Create VI Interface to CVI Instrument Driver tool available from the Instrument Driver Network at ni.com/idnet. You can call IVI-COM instrument class drivers and specific drivers from any development environment that supports ActiveX. Use the ActiveX/COM Adapter to configure steps to access objects IVI-COM class instrument drivers define.

Refer to the *Plug and Play Instrument Drivers* section of this appendix for more information about LabVIEW and LabWindows/CVI Plug and Play drivers.

Use the IVI step types to configure and acquire data from IVI class instruments. Use an initial IVI step to configure an instrument and use one or more subsequent IVI steps to perform measurements. TestStand uses the instrument logical name to reference a session to an instrument. Use National Instruments Measurement & Automation Explorer (MAX) to configure instrument logical names. TestStand initializes the instrument session when you first configure the instrument and closes the instrument

session when the execution closes. If two executions reference the same logical name, TestStand shares the session, and the session closes when TestStand releases the last execution of the two.

IVI step types use the National Instruments Session Manager to share named instrument connections. You can also use Session Manager to share instrument connections in code modules even if you do not use IVI step types. Access the *NI Session Manager Help* by selecting **Start»All Programs»National Instruments»Session Manager»NI Session Manager Help**.



Note With IVI-C drivers, you cannot use the same instrument driver session in more than one operating system process simultaneously.

Although you can use IVI step types to configure and acquire data from IVI class instruments, you must use code modules to control instruments to ensure optimal performance by precisely specifying the instrument driver calls, to call specific driver functions an IVI class does not support, to interleave instrument control operations with other code that must reside in a single code module, when the instrument does not conform to an IVI class, or when no IVI driver exists for the instrument.

Right-click an IVI step and select **Edit <step type>** from the context menu to configure and select an operation for the step to perform. You can also click the **Edit <step type>** button on the edit tab of the Step Settings pane. Refer to the *NI TestStand Help* for more information about each Edit IVI <Step Name> dialog box and for information about the operation each IVI step type can perform.

When TestStand configures an instrument, the instrument driver might coerce a settings value. Configuring an instrument in TestStand might result in an error that indicates an invalid value for a particular setting because TestStand does not validate the instrument-based values until the configuration actually occurs. When you edit a step that configures an instrument, click the **Validate** button in the Edit IVI <Step Name> dialog box to test the configuration before you close the dialog box.

Use the instrument soft front panel (SFP), which is a graphical display panel for the instrument, to interact directly with the instrument session TestStand controls.

Refer to the <TestStand Public>\Examples\IVI directory for examples of how to use the IVI step types.

IVI Dmm



Use the IVI Dmm step, shown at left, to perform single-point and multipoint measurements with digital multimeters.

Step Operations

The IVI Dmm step type supports the following operations:

- **Configure**—Configures the instrument to match the state the step specifies.
- **Show Soft Front Panel**—Launches the SFP for the instrument.
- **Hide Soft Front Panel**—Hides the SFP for the instrument.
- **Read**—Initiates and returns a measurement from an instrument.
- **Initiate**—Initiates a measurement from an instrument.
- **Fetch**—Returns the measured value from a measurement the Initiate operation started.
- **Abort**—Cancels the wait for a trigger.
- **Send Software Trigger**—Sends a software command to trigger the instrument.
- **Get Information**—Retrieves low-level status and information from the instrument.

Step Properties

In addition to the common custom properties, the IVI Dmm step type defines the following step properties:

- **Step.Result.Reading**—Contains the measurement values for the Read and Fetch operations. The property data type is NI_IviSinglePoint or NI_IviWave.
- **Step.LogicalName**—Contains the logical name expression.
- **Step.InstrOperation**—Contains a value that specifies the operation you configured the step to perform.
- **Step.SettingsSource**—Contains the name of the property or variable where the step loads and stores the settings for the operation.
- **Step.Configuration**—Contains the settings for the Configure operation. The data type of this property is NI_IviDmmConfig.
- **Step.SoftFrontPanel**—Contains the settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.

- **Step.Readings**—Contains the settings for the Read and Fetch operations.
- **Step.GetInfo**—Contains the settings for the Get Information operation.

IVI Scope



Use the IVI Scope step, shown at left, to acquire a voltage waveform from an analog input signal using oscilloscopes.

Step Operations

The IVI Scope step type supports the following operations:

- **Configure**—Configures the instrument to match the state the step specifies.
- **Show Soft Front Panel**—Launches the SFP for the instrument.
- **Hide Soft Front Panel**—Hides the SFP for the instrument.
- **Read**—Initiates and returns a measurement from an instrument.
- **Initiate**—Initiates a measurement from an instrument.
- **Fetch**—Returns the measured value from a measurement the Initiate operation started.
- **Abort**—Cancels an ongoing Initiate operation.
- **Auto Setup**—Performs an automatic setup on the instrument.
- **Get Information**—Retrieves low-level status and information from the instrument.

Step Properties

In addition to the common custom properties, the IVI Scope step type defines the following step properties:

- **Step.Result.Reading**—Contains the measurement values for the Read and Fetch operations. This property is a container array, and the size of the array equals the number of channels you specify for the Read or Fetch operation. The data type of each element of the array is NI_IviSinglePoint, NI_IviWave, or NI_IviWavePair.
- **Step.LogicalName**—Contains the logical name expression.
- **Step.InstrOperation**—Contains a value that specifies the operation you configured the step to perform.

- **Step.SettingsSource**—Contains the name of the property or variable where the step loads and stores the settings for the operation.
- **Step.Configuration**—Contains the settings for the Configure operation. The data type of this property is NI_IviScopeConfig.
- **Step.SoftFrontPanel**—Contains the settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.
- **Step.Readings**—Contains the settings for the Read and Fetch operations. The data type of this property is NI_IviScopeReadings. The Channels subproperty is an NI_IviScopeChannel array.
- **Step.GetInfo**—Contains the settings for the Get Information operation.

IVI Fgen



Use the IVI Fgen step, shown at left, to instruct function generators to generate predefined and custom waveforms using arbitrary waveform generators.

Step Operations

The IVI Fgen step type supports the following operations:

- **Configure**—Configures the instrument to match the state the step specifies.
- **Show Soft Front Panel**—Launches the SFP for the instrument.
- **Hide Soft Front Panel**—Hides the SFP for the instrument.
- **Initiate**—Initiates signal generation if the instrument is idle.
- **Abort**—Aborts a previously configured output and returns the function generator to the idle state.
- **Send Software Trigger**—Sends a software command to trigger the instrument.
- **Get Information**—Retrieves low-level status and information from the instrument.

Step Properties

In addition to the common custom properties, the IVI Fgen step type defines the following step properties:

- **Step.LogicalName**—Contains the logical name expression.
- **Step.InstrOperation**—Contains a value that specifies the operation you configured the step to perform.
- **Step.SettingsSource**—Contains the name of the property or variable where the step loads and stores the settings for the operation.
- **Step.Configuration**—Contains the settings for the Configure operation. The data type of this property is NI_IviFgenConfig.
- **Step.SoftFrontPanel**—Contains the settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.
- **Step.GetInfo**—Contains the settings for the Get Information operation.

IVI Power Supply



Use the IVI Power Supply step, shown at left, to instruct DC power supplies to control the output voltages and currents and to measure output values at the output terminals.

Step Operations

The IVI Power Supply step type supports the following operations:

- **Configure**—Configures the instrument to match the state the step specifies.
- **Show Soft Front Panel**—Launches the SFP for the instrument.
- **Hide Soft Front Panel**—Hides the SFP for the instrument.
- **Measure**—Takes a measurement on the output signal and returns the measured value.
- **Initiate**—Makes the power supply wait for a trigger.
- **Abort**—Cancels the wait for a trigger.
- **Send Software Trigger**—Sends a software command to trigger the instrument.

- **Reset Output Protection**—Resets the output protection of the power supply on a specific channel after an overvoltage or overcurrent condition occurs.
- **Get Information**—Retrieves low-level status and information from the instrument.

Step Properties

In addition to the common custom properties, the IVI Power Supply step type defines the following step properties:

- **Step.Result.Reading**—Contains the measurement values for the Measure operation. The property data type is an array of `NI_IviSinglePoint`.
- **Step.LogicalName**—Contains the logical name expression.
- **Step.InstrOperation**—Contains a value that specifies the operation you configured the step to perform.
- **Step.SettingsSource**—Contains the name of the property or variable where the step loads and stores the settings for the operation.
- **Step.Configuration**—Contains the settings for the Configure operation. The data type of this property is `NI_IviDCPowerConfig`.
- **Step.SoftFrontPanel**—Contains the settings for the Show Soft Front Panel operation. The data type of this property is `NI_IviSoftFrontPanel`.
- **Step.Readings**—Contains the settings for the Measure operation. The data type of this property is `NI_IviDCPowerReadings`.
- **Step.GetInfo**—Contains the settings for the Get Information operation.
- **Step.ResetOutputProtection**—Contains the channel setting for the Reset Output Protection operation.

IVI Switch



The IVI Switch step, shown at left, provides a high-level programming layer for instruments compliant with the IVI Switch class and NI Switch Executive virtual devices. A switch is an instrument that can establish a connection between two I/O channels. The IVI Switch step type also supports IVI-compliant instruments that can perform trigger scanning and trigger-synchronized path connection and disconnection.

NI Switch Executive is an intelligent switch management and routing application you can use with TestStand to interactively configure switch devices from multiple vendors as a single virtual device. You can specify intuitive names for each channel within the virtual switch device and use an end-to-end routing feature to automatically find switch routes by selecting the channels you want to connect. Refer to ni.com/switchexecutive for more information about NI Switch Executive.

Use the IVI Switch step type to connect and disconnect paths and routes, to determine the connectivity of two switches or the state of a route, and to query the state of the switch module or virtual device. Use the Switching panel on the Properties tab of the Step Settings pane to connect and disconnect routes required for steps in sequences. When you install NI Switch Executive, you can also use the Switching panel to specify a switching action TestStand performs around the execution of the step. Refer to the *NI TestStand Help* for more information about the Switching panel.

Route Specification String

When you instruct TestStand to connect or disconnect routes you define in an NI Switch Executive virtual device, you must specify a route specification string. NI Switch Executive ignores whitespace characters between tokens in a route specification string.

The syntax of a route specification string consists of an ampersand-delimited series of routes, as shown in the following example:

```
routeOrGroup { & routeOrGroup } { & routeOrGroup } . . .
```

where `routeOrGroup` is a route name, a route group name, or a fully specified path enclosed in square brackets and consisting of a series of channels delimited by "`->`", as shown in the following example:

```
[ channel {-> channel } {-> channel} . . . ]
```

where `channel` is a channel alias name, an IVI channel name, or a unique name, which is a combination of the IVI device logical name and the IVI channel name separated by a "/" delimiter.

Channels on each end of a bracketed, fully specified path must not be Configuration or Hardwired channels. Only one end channel can be a Source channel. The inner channels in a route specification string must be Configuration or Hardwired channels.

The following example is a complete route specification string:

```
MyRouteGroup & MyRoute & [Dev1/CH3->CH4,CH4->R0]
```

Step Operations

The IVI Switch step type supports the following IVI Switch class operations:

- **Connect/Disconnect**—Connects or disconnects the Source and Destination channels in the switch instrument.
- **Configure Scan**—Configures the switch instrument for scanning.
- **Start Scan**—Initiates a scanning operation.
- **Wait**—Blocks operations until all switches debounce for an instrument.
- **Configure Switch**—Configures channels as Configuration or Source channels and configures specific paths between channels.
- **Send Software Trigger**—Sends a software command to trigger the instrument during a scanning operation.
- **Abort Scan**—Cancels a scanning operation.
- **Get Information**—Retrieves low-level status and information from the instrument.

The IVI Switch step type supports the following NI Switch Executive operations:

- **Connect/Disconnect**—Connects or disconnects switch routes for a virtual device.
- **Wait**—Blocks operations until all switches debounce for a virtual device.
- **Get Information**—Retrieves low-level status and information from a virtual device.

Step Properties

In addition to the common custom properties, the IVI Switch step type defines the following step properties:

- **Step.LogicalName**—Contains the logical name expression.
- **Step.InstrOperation**—Contains a value that specifies the operation you configured the step to perform.
- **Step.SettingsSource**—Contains the name of the property or variable where the step loads and stores the settings for the operation.
- **Step.IviOperation**—Contains a value that specifies the operation you configured the step to perform for IVI Switching mode.
- **Step.ConnectDisconnect**—Contains the settings for the Connect/Disconnect operation.
- **Step.SoftFrontPanel**—Contains the settings for the Show Soft Front Panel operation. The data type of this property is `NI_IviSoftFrontPanel`.
- **Step.GetInfo**—Contains the settings for the Get Information operation.
- **Step.ScanningConfig**—Contains the settings for the Configure Scan operation.
- **Step.Wait**—Contains the settings for the Wait operation.
- **Step.Configure**—Contains the settings for the Configure operation.

IVI Tools



Use the IVI Tools step, shown at left, to perform low-level operations on an instrument.

Step Operations

The IVI Tools step type supports the following operations:

- **Get Session Info**—Retrieves low-level session references and API class handles to the IVI instrument.
- **Show Soft Front Panel**—Launches the SFP for the tool.
- **Hide Soft Front Panel**—Hides the SFP for the tool.
- **Init**—Initializes the driver or I/O resource for the session.
- **Close**—Closes the IVI session.
- **Reset**—Places the instrument in a known state.

- **Self Test**—Causes the instrument to perform a self-test.
- **Revision Query**—Queries the instrument driver and instrument for revision information.
- **Error Query**—Returns instrument-specific error information.
- **Get Error Info**—Returns error information for the last IVI error in a session.
- **Set/Get/Check Attributes**—Sets, queries, or verifies the value of attributes.

Step Properties

In addition to the common custom properties, the IVI Tools step type defines the following step properties:

- **Step.LogicalName**—Contains the logical name expression.
- **Step.InstrOperation**—Contains a value that specifies the operation you configured the step to perform.
- **Step.SettingsSource**—Contains the name of the property or variable where the step loads and stores the settings for the operation.
- **Step.SoftFrontPanel**—Contains the settings for the Show Soft Front Panel operation. The data type of this property is `NI_IviSoftFrontPanel`.
- **Step.Init**—Contains the settings for the Init operation.
- **Step.SelfTest**—Contains the settings for the Self Test operation.
- **Step.SessionInfo**—Contains the settings for the Get Session Info operation.
- **Step.RevisionQuery**—Contains the settings for the Revision Query operation.
- **Step.ErrorQuery**—Contains the settings for the Error Query operation.
- **Step.ErrorInfo**—Contains the settings for the Get Error Info operation.
- **Step.Attributes**—Contains the settings for the Set/Get/Check Attributes operation.

Plug and Play Instrument Drivers

Plug and Play drivers simplify controlling and communicating with the instrument through a standard, straightforward programming model for all drivers. Plug and Play drivers exist for LabVIEW and LabWindows/CVI.

A LabVIEW Plug and Play instrument driver is a set of VIs. Each VI corresponds to a programmatic operation for the instrument. National Instruments distributes LabVIEW Plug and Play instrument drivers with the block diagram source code so you can customize the VIs. You can create instrument control applications and systems by programmatically linking instrument driver VIs on the block diagram. LabVIEW Plug and Play instrument drivers usually use VISA functions to communicate with instruments.

In TestStand, you can call VIs that use LabVIEW Plug and Play instrument drivers. When you return a VISA reference to TestStand and later pass the reference to a different VI code module that uses the same instrument driver, store the reference in a TestStand LabVIEWIORreference variable. You can also use the LabVIEW Adapter to directly call VIs in an instrument driver.

A LabWindows/CVI Plug and Play instrument driver is a set of ANSI C software routines exported from a DLL. You can call these instrument drivers from any development environment that supports calls into DLLs. You can also use a LabWindows/CVI instrument driver in LabVIEW if you convert the instrument driver using the Create VI Interface to CVI Instrument Driver tool available from the Instrument Driver Network at ni.com/idnet. LabWindows/CVI Plug and Play instrument drivers are based on the *VXIplug&play* standard architecture and usually use VISA functions to communicate with instruments.

In TestStand, you can call code modules that use LabWindows/CVI Plug and Play instrument drivers. When you return a C-based reference to TestStand and later pass the reference to a different code module that uses the same instrument driver, store the reference in a TestStand numeric variable. You can also use the LabWindows/CVI or C/C++ DLL Adapter to directly call the functions in an instrument driver.

LabVIEW Utility Step Types



Use the LabVIEW Utility step types to simplify running a VI on a remote computer and to deploy or undeploy shared variables. All the LabVIEW Utility step types use the icon shown at left.

Check Remote System Status

Use the Check Remote System Status step type to determine if LabVIEW is running on a remote computer and if TestStand can connect to the remote computer.

Step Properties

In addition to the common custom properties, the Check Remote System Status step type defines the following step properties:

- **Step.RemoteHost**—Specifies the remote computer name or IP address.
- **Step.RemoteHostByExpr**—Specifies if you can use an expression in the Hostname field.
- **Step.PortNumber**—Specifies the remote host port number.
- **Step.Timeout**—Specifies the number of seconds to wait to connect to the remote computer.
- **Step.ServerCheckExpr**—Specifies where to store a Boolean value that indicates if the remote computer check passed.

Run VI Asynchronously

Use the Run VI Asynchronously step type to run a VI in a new thread in the TestStand execution.

Step Properties

In addition to the common custom properties, the Run VI Asynchronously step type defines the following step properties:

- **Step.RemoteHost**—Specifies the remote computer name or IP address.
- **Step.RemoteHostByExpr**—Specifies if you can use an expression in the Hostname field.
- **Step.PortNumber**—Specifies the remote host port number.
- **Step.Timeout**—Specifies the number of seconds to wait to connect to the remote computer.
- **Step.VIModule**—Contains the settings for the VI the step calls.

Deploy Library

Use the Deploy Library step type to deploy or undeploy shared variables defined in a LabVIEW project library file to or from the target computer. Refer to the *Network-Published Shared Variables* section of Appendix A, *Using LabVIEW 8.x with TestStand*, of the *Using LabVIEW with TestStand* manual for more information about shared variables.

Step Properties

In addition to the common custom properties, the Deploy Library step type defines the following step properties:

- **Step.Operation**—Specifies if the step deploys or undeploys shared variables. Valid values are 0 = Deploy and 1 = Undeploy.
- **Step.Libraries**—Specifies an expression for the path of the project library file on the local computer to deploy to or undeploy from the target computer. The project library must define only shared variables and cannot contain any VI files.

Technical Support and Professional Services

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

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

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

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

Index

A

- Action step, 4-15
- ActiveX controls
 - interface pointer, obtaining, 9-17
 - Multiple Document Interface (MDI)
 - application, creating, 9-15
 - using with DLLs, 5-5
 - using with Microsoft Visual C++, 9-16
- ActiveX/COM Adapter, 1-7, 5-12
 - compatibility options for Visual Basic, 5-13
 - registering and unregistering servers, 5-13
- ActiveX/COM server
 - compatibility options, 5-13
 - compatibility options for Visual Basic, 5-13
 - debugging, 5-13
 - registering and unregistering, 5-13
- adapters. *See* module adapters
- Additional Results panel, 4-5
- Additional Results step, 4-25
- Application Manager control, 9-3
 - command-line arguments, 9-29
 - generating events, 9-18
- application manifests, 9-33
- application settings
 - configuration file location, 9-30
 - custom, adding, 9-31
 - persisting, 9-30
- application styles
 - multiple window, 9-27
 - no visible window, 9-29
- array property, 1-8
- arrays
 - dynamic array sizing, 12-3
 - empty, 12-4
 - specifying array sizes, 12-2

- Authenticode signatures, 9-32
- Auto Schedule step, B-15
- automatic result collection, 1-16

B

- Batch process model, A-5, A-20
 - Configuration entry point, A-24
 - Engine callbacks, A-27
 - hidden Execution entry points, A-24
 - Model callbacks, A-24
 - Single Pass – Test Socket entry point, A-32
 - Single Pass entry point, A-31
 - Test UUTs – Test Socket entry point, A-29
 - Test UUTs entry point, A-27
 - utility subsequence, A-27
- Batch reports, 6-15
- Batch Specification step, B-19
- Batch Synchronization step, B-13
 - mismatched sections, B-14
 - nested sections, B-14
 - One Thread Only section, B-14
 - Parallel section, B-14
 - Serial section, B-13
 - step properties, B-15
 - synchronized sections, B-13
- Break step, 4-19
- built-in step properties, 4-3
- built-in step types
 - Action, 4-15
 - Additional Results, 4-25
 - Call Executable, 4-23
 - custom properties, 4-6
 - FTP Files, 4-24
 - Label, 4-20
 - Message Popup, 4-21
 - module adapters, using with, 4-7

- Multiple Numeric Limit Test, 4-11
- Numeric Limit Test, 4-9
- Pass/Fail Test, 4-8
- Property Loader. *See* Property Loader step
- Sequence Call, 2-5, 4-15
- Statement, 4-20
- String Value Test, 4-13
- Button control
 - description (table), 9-5

C

- C++ (MFC)
 - creating event handlers (table), 9-18
 - localization functions (table), 9-26
 - menu open notification methods (table), 9-24
 - using the TSUtil library (table), 9-22
- C/C++ DLL Adapter, 1-7, 5-4
- Call Executable step, 4-23
- Call Stack pane, 3-5
- callbacks
 - Engine, 1-16, 3-13, 10-4, A-10
 - caveats, 10-8
 - predefined (note), 10-4
 - table, 10-5
 - Front-End, 10-9
 - modifying, 10-9
 - Model, 1-16
 - Batch process model, A-24
 - Parallel process model, A-16
 - Sequential process model, A-7
 - modifying, 10-4
 - sequence executions, 1-17
 - sequence file callbacks, 2-2
 - sequences, 10-2
 - types (table), 1-17
- caption connection, 9-10
- Case step, 4-19
- Check Remote System Status step, E-1
- CheckBox control
 - description (table), 9-5
- client sequence file, 1-15
- Close Database step, C-3
- Close SQL Statement step, C-3
- code modules, 1-1
 - dynamic array sizing, 12-3
 - parameterized, creating, 1-13
 - parameters, accessing from, 1-13
 - specifying, 4-7
 - verifying privileges, 7-2
- code templates, 13-6
 - Code Templates tab, 13-6
 - Create Code Templates dialog box, 13-8
 - creating, 13-8
 - customizing, 13-8
 - default code templates (table), 13-6
 - development environments, 13-6
 - legacy, 13-7
 - locating with search paths
 - <TestStand Public> subdirectory (table), 8-5
 - <TestStand> subdirectory (table), 8-3
 - module adapters, 5-2
 - multiple, specifying, 13-8
 - source code templates, 1-11
 - specifying multiple code templates, 13-8
- collecting results, 1-16, 3-7
- ComboBox control
 - description (table), 9-5
- command connections, 9-9
 - invoking, 9-10
- command-line arguments, 9-29
- configuration
 - See also* customizing TestStand
 - configuration file location,
 - user interface, 9-30
 - Configure menu, 8-11
 - module adapters, 5-1

- remote sequence execution, 5-19
 - Windows 2000 SP4, 5-22
 - Windows Vista, 5-20
 - Windows XP SP2, 5-20
- sequence editor or user interface startup
 - options (table), 8-9
- Configuration entry point, 1-15, 10-4
 - Database Options, 10-4
 - Model Options, 10-4
 - process models, A-4
 - Batch process model, A-24
 - Parallel process model, A-16
 - Sequential process model, A-6
 - Report Options, 10-4
- configuration file location, user interface, 9-30
- Configure Database Options entry point, A-7
- Configure menu, 8-11
- Configure Model Options entry point, A-7
- Configure Report Options entry point, A-6
- connections
 - command, 9-9
 - invoking, 9-10
 - information source
 - caption, 9-10
 - image, 9-11
 - numeric value, 9-12
 - list (table), 9-8
- container property, 1-8
- Continue step, 4-19
- custom application settings, adding, 9-31
- custom data types, 1-9, 1-10
 - creating and modifying, 12-7
- custom properties
 - See also* step properties
 - built-in step types, 4-6
 - lifetime of, 3-3
 - result properties, 3-8
 - step properties, 1-10
- custom sequence files
 - deploying, 15-4
 - versioning, 15-3

- custom step types
 - See also* step types
 - creating, 13-1
 - step templates, differences with, 13-1
- custom user interfaces
 - documenting, 9-31
- custom user privileges, 7-3
- customizing TestStand
 - components (table), 8-1
 - Tools menu, 8-2

D

- data links, 6-4, 6-11
 - specifying (tutorial), 6-12
- Data Operation step, C-4
- data source, 4-12
- data types
 - categories, 12-2
 - common properties, 12-7
 - creating custom data types (tutorial), 12-7
 - creating instances from context menus
 - (table), 12-1
 - custom properties, 12-8
 - displaying, 12-4
 - modifying, 12-4
 - named, 12-2
 - simple, 12-2
 - standard, 12-5
 - CommonResults, 12-6
 - Error, 12-6
 - Path, 12-6
 - using, 12-5
 - standard named, 1-9
- Database step types
 - Close Database, C-3
 - Close SQL Statement, C-3
 - Data Operation, C-4
 - Open Database, C-1
 - Open SQL Statement, C-2

- Property Loader, C-5
 - loading from database, C-7
 - loading from file, C-6
- database viewer. *See* databases
- databases
 - adding support for database management systems, 6-9
 - concepts, 6-1
 - connection strings, 6-4
 - data links, 6-4
 - database technologies
 - Microsoft Object Linking and Embedding Database (OLE DB), 6-2
 - Microsoft Open Database Connectivity (ODBC), 6-2
 - table, 6-3
- example (table), 6-1
- logging property in the sequence context, 6-7
- logging results, 6-5
 - configuring options (tutorial), 6-12
 - on-the-fly, 6-11
 - using the default process model, 6-6
- result tables, 6-8
 - creating (tutorial), 6-13
 - creating with TestStand Database Viewer, 6-9
 - default TestStand table schema, 6-8
- sessions, 6-2
- specifying data links, 6-12
- specifying schemas, 6-12
- step types. *See* Database step types
- TestStand Database Viewer
 - result tables, creating (tutorial), 6-13
- DCOM. *See* Distributed COM (DCOM)
- debug
 - DLLs, 5-7
 - HTBasic Adapter, 5-15
 - .NET assemblies, 5-9
 - panes, 3-5
- Deploy Library step, E-2
- deployment. *See* TestStand Deployment Utility
- diagnostic tools (NI resources), F-1
- directory structure, 8-2
 - <TestStand Application Data> directory, 8-7
 - <TestStand Public> directory, 8-5
 - RuntimeServers directory, 8-6
 - subdirectories (table), 8-5
 - <TestStand> directory, 8-2
 - Components directory, 8-3
 - subdirectories (table), 8-3
 - subdirectories (table), 8-3
 - read-only files, copying to modify, 8-6
 - search paths, 5-2
- DisplaySequenceFile event, 9-19
- Distributed COM (DCOM), 5-18
 - configuring
 - Windows Vista, 5-18
 - Windows XP SP2, 5-18
- DLLs
 - ActiveX controls, using, 5-5
 - debugging, 5-7
 - DLL functions, options for stepping out of (table), 5-8
 - LabVIEW 7.1.1 shared libraries, 5-8
 - LabVIEW 8 or later, 5-5
 - LabVIEW 8.0 and later shared libraries, 5-8
 - Microsoft Foundation Class (MFC) Library, using, 5-5
 - parameter information, reading, 5-7
 - subordinate, loading, 5-6
 - translator DLL, creating, 15-2
 - using with TestStand, 5-5
- Do While step, 4-18

drivers

- LabVIEW Plug and Play, D-12
- LabWindows/CVI Plug and Play, D-12
- NI resources, F-1

dynamic array sizing, 12-3

E

Edit Search Directories dialog box, 5-2

editing sequence files

- callback sequences, 10-2
- entry point sequences, 10-3
- normal sequences, 10-2

Editor applications, creating, 9-13

Else If step, 4-17

Else step, 4-16

empty arrays, 12-4

End step, 4-20

Engine callbacks, 1-16, 3-13, 10-4, A-10

- Batch process model, A-27
- caveats, 10-8
- Parallel process model, A-17
- predefined (note), 10-4
- Sequential process model, A-10
- table, 10-5

engine. *See* TestStand Engine

entry points

- Configuration, 1-15
- entry point sequences, 10-3
- Execution, 1-15, 3-4

errors

- run-time, 1-12, 3-18
 - caution, 4-4
 - handling interactively, 3-18
- step failure, 3-17

event handling, 9-17

- DisplayExecution event, 9-19
- ExitApplication event, 9-18
- ReportError event, 9-19
- shut down, 9-20
- startup, 9-20

Wait event, 9-18

examples (NI resources), F-1

Execution entry point, 1-15

- Parallel process model, A-13
- Sequential process model, A-6
- Single Pass, 10-3, A-4

Batch process model, A-31

Parallel process model, A-19

Sequential process model, A-12

Single Pass – Test Socket entry point

Batch process model, A-24, A-32

Parallel process model, A-15, A-19

Test UUTs, 10-3, A-4

Batch process model, A-27

Parallel process model, A-17

Sequential process model, A-11

Test UUTs – Test Socket entry point

Batch process model, A-24, A-29

Parallel process model, A-15, A-18

using Execution entry points, 3-4

Execution object, 1-17

execution pointer, 3-1

Execution window, 3-3

executions, 3-1

- aborting, 3-6, 3-7
- debugging, 3-5
 - Call Stack pane, 3-5
 - Output pane, 3-6
 - Threads pane, 3-5
 - Variables pane, 3-6
 - Watch View pane, 3-6

direct, 3-4

executing sequences, 3-4

execution pointer, 3-1

Execution window, 3-3

interactive, 3-5

order of execution (table), 3-14

remote sequence execution, 5-17

run-time copy (note), 3-1

run-time errors, 3-18

- step execution, 3-14
- step status, 3-16
- terminating, 3-6, 3-18
- using Execution entry points, 3-4

ExecutionView Manager control, 9-4

- connecting views, 9-7

ExpressionEdit control

- description (table), 9-6

expressions, 1-10

- context-sensitive editing, 1-10
- sequence context, using, 3-2

Expressions panel, 4-5

F

file types

- configuration file location, 9-30
- project, 2-5
- string resource files, creating, 8-7
- type palette, 11-4
- workspace, 2-5

files

- collecting, 14-3
- read-only, copying to modify, 8-6

Flow Control step types, 4-16

- Break, 4-19
- Case, 4-19
- Continue, 4-19
- Do While, 4-18
- Else, 4-16
- Else If, 4-17
- End, 4-20
- For, 4-17
- For Each, 4-18
- Goto, 4-20
- If, 4-16
- Select, 4-19
- While, 4-18

For Each step, 4-18

For step, 4-17

Front-End callbacks, 10-9

- modifying, 10-9

FTP Files step, 4-24

G

General panel, 4-3

Goto step, 4-20

H

hidden Execution entry points

- Batch process model, A-24
- Parallel process model, A-15

HTBasic Adapter, 1-7, 5-15

- debugging subroutines, 5-15

I

If step, 4-16

image connections, 9-11

information source connection

- caption, 9-10
- image, 9-11
- numeric value, 9-12

Insert Step submenu, 4-1

Insertion Palette, 4-1

- figure, 2-4, 4-2

InsertionPalette control

- description (table), 9-6

Instrument Driver Network, D-1

instrument drivers, D-1

- NI resources, F-1
- Plug and Play, D-12
- LabVIEW, D-12
- LabWindows/CVI, D-12

instrument logical name, D-1

instrument session, D-1

instrument soft front panel, D-2

interactive execution, 3-5

- interactive mode, 3-5

interface pointer, obtaining, 9-17

IVI, D-1

instrument drivers. *See* instrument drivers

IVI-C, D-1

IVI-COM, D-1

soft front panel, D-2

step types. *See* IVI step types

steps, editing, D-2

IVI step types

code modules, using instead, D-2

IVI Dmm, D-3

IVI Fgen, D-5

IVI Power Supply, D-6

IVI Scope, D-4

IVI Switch, D-8

IVI Tools, D-10

K

KnowledgeBase, F-1

L

Label control

description (table), 9-6

Label step, 4-20

LabVIEW

creating event handlers (table), 9-18

localization functions (table), 9-26

menu open notification methods
(table), 9-24

module adapter, 1-7

Plug and Play drivers, D-12

using TestStand User Interface (UI)
Controls, 9-14

using TSUtil library (table), 9-22

Utility step types

Check Remote System Status, E-1

Deploy Library, E-2

Run VI Asynchronously, E-1

LabVIEW Adapter, 1-7, 5-4

LabWindows/CVI

creating event handlers (table), 9-18

localization functions (table), 9-26

menu open notification methods
(table), 9-24

module adapter, 1-7

Plug and Play drivers, D-12

using TestStand User Interface (UI)
Controls, 9-14

using TSUtil library (table), 9-22

LabWindows/CVI Adapter, 5-4

creating event handlers (table), 9-18

localization functions (table), 9-26

legacy code template, 13-7

license checking, 9-13

lifetime

custom step properties, 3-3

local variables, 3-3

parameters, 3-3

Synchronization step types, B-4

list connections (table), 9-8

ListBar control

description (table), 9-6

ListBox control

description (table), 9-6

local variables, 2-4

lifetime, 3-3

sequence local, 1-12

localization, 9-25

Lock step, B-5

logging results

database, on-the-fly, 6-11

loop results, 3-13

Looping panel, 4-4

M

- Main sequence, 1-15
- Manager controls
 - Application Manager, 9-3
 - ExecutionView Manager, 9-4
 - SequenceFileView Manager, 9-4
- manifests. *See* application manifests
- Measurement & Automation Explorer (MAX), D-1
- menus
 - Configure, 8-11
 - Tools menu
 - customizing, 8-2
 - modifying, 8-2
 - updating, 9-24
- merging sequence files, 2-2
- Message Popup step, 4-21
- Microsoft
 - Access
 - example data link and result table setup, 6-12
 - specifying data link and schema, 6-12
 - ActiveX Data Objects (ADO), 6-2
 - Microsoft Foundation Class (MFC)
 - Library
 - using in DLLs, 5-5
 - Object Linking and Embedding Database (OLE DB), 6-2
 - Visual Basic
 - compatibility options with
 - ActiveX/COM server, 5-13
 - Visual C++
 - using TestStand User Interface (UI) Controls, 9-16
 - Visual Studio
 - using TestStand User Interface (UI) Controls, 9-15
 - Visual Studio .NET 2003
 - assembly references, adding, 9-23
 - .NET assemblies, debugging, 5-9
 - TestStand API, accessing, 5-12
 - Visual Studio 2005
 - assembly references, adding, 9-23
 - Model callbacks, 1-16, 10-2
 - Batch process model, A-24
 - Parallel process model, A-16
 - report generation (table), A-38
 - Sequential process model, A-7
 - model sequence files, 2-1
 - modifying types, 11-1
 - module adapters, 1-7
 - ActiveX/COM, 1-7, 5-12
 - registering and unregistering servers, 5-13
 - using ActiveX/COM servers with TestStand, 5-13
 - C/C++ DLL, 1-7, 5-4
 - configuring, 5-1
 - HTBasic, 1-7, 5-15
 - LabVIEW, 1-7, 5-4
 - LabWindows/CVI, 1-7, 5-4
 - .NET, 1-7, 5-9
 - overview, 1-7
 - Sequence, 1-7, 4-15, 5-16
 - source code templates, 5-2
 - supported code modules, 5-1
 - using built-in step types, 4-7
 - monitoring variables, 3-2
 - Multiple Document Interface (MDI)
 - application, creating, 9-15
 - Multiple Numeric Limit Test step, 4-11
 - multithreading, 1-17

N

- National Instruments support and services, F-1
- .NET
 - creating event handlers (table), 9-18
 - localization functions (table), 9-26
 - menu open notification methods (table), 9-24
 - using TSUtil library (table), 9-22

- .NET Adapter, 1-7, 5-9
 - .NET assemblies, debugging, 5-9
- NI support and services, F-1
- NI Switch Executive, D-8
 - route specification string, D-8
- normal sequence files, 2-1
- normal sequences, 10-2
- Notification step, B-10
- Numeric Limit Test step, 4-9
- numeric value connection, 9-12

O

- object
 - Execution, 1-17
 - SequenceContext, 1-9
 - Synchronization, B-1
 - common attributes, B-3
 - User, 7-3
- Object Linking and Embedding Database (OLE DB)
 - data links, using, 6-11
- object reference properties, 12-5
- Open Database Connectivity (ODBC), 6-2
 - Administrator, using, 6-12
 - data links, using, 6-11
- Open Database step, C-1
- Open SQL Statement step, C-2
- Operator Interface application, 9-13
- operator interface. *See* user interfaces
- Output pane, 3-6

P

- panels
 - Additional Results, 4-5
 - Expression, 4-5
 - Expressions, 4-5
 - General, 4-3
 - Looping, 4-4
 - Post Actions, 4-4
 - Preconditions, 4-5
 - Property Browser, 4-5
 - Requirements, 4-5
 - Run Options, 4-3
 - Switching, 4-4
 - Synchronization, 4-4
- panes
 - Call Stack, 3-5
 - debugging, 3-5
 - Output, 3-6
 - Sequence File window, 2-5
 - Sequences pane, 2-5
 - Steps pane, 2-5
 - Variables pane, 2-5
 - Step Settings. *See* Step Settings pane
 - Steps, 2-3, 2-5
 - TestStand Sequence Editor
 - Workspace pane, 2-6
 - Threads, 3-5
 - Types, 2-2
 - Variable, 3-6
 - Variables, 2-2, 2-3, 2-4, 12-4
 - View Types For, 11-4
 - Watch View, 3-6
 - Workspace, 2-6
- Parallel process model, A-5, A-13
 - Configuration entry point, A-16
 - Engine callbacks, A-17
 - Execution entry points, A-13
 - hidden, A-15
 - Model callbacks, A-16
 - Single Pass – Test Socket
 - entry point, A-19
 - Single Pass entry point, A-19
 - Test UUTs – Test Socket
 - entry point, A-18
 - Test UUTs entry point, A-17
 - utility sequences, A-13

- parameters
 - accessing from code modules, 1-13
 - complex data types, 12-2
 - lifetime, 3-3
 - reading, 5-7
 - sequence, 1-13, 2-3
- Pass/Fail Test step, 4-8
- Plug and Play instrument drivers
 - LabVIEW, D-12
 - LabWindows/CVI, D-12
- Post Actions panel, 4-4
- Preconditions panel, 4-5
- privileges
 - See also* user privileges
 - accessing privilege settings
 - any user, 7-3
 - current user, 7-2
 - defining custom privileges, 7-3
- process models, 1-14
 - architecture, A-1
 - Batch, A-5, A-20
 - Configuration entry point, A-24
 - Engine callbacks, A-27
 - hidden Execution entry point, A-24
 - Model callbacks, A-24
 - Single Pass – Test Socket entry point, A-32
 - Single Pass entry point, A-31
 - Test UUTs – Test Socket entry point, A-29
 - Test UUTs entry point, A-27
 - utility sequences, A-21
 - common features, A-4
 - Configuration entry point, A-4
 - entry points, 1-15
 - Execution entry point, A-4
 - modifying process model
 - sequence files, 10-1
 - Parallel, A-5
 - Configuration entry point, A-16
 - Engine callbacks, A-17
 - Execution entry points, A-13
 - hidden Execution entry points, A-15
 - Model callbacks, A-16
 - Single Pass – Test Socket entry point, A-19
 - Single Pass entry point, A-19
 - Test UUTs – Test Socket entry point, A-18
 - Test UUTs entry point, A-17
 - utility sequences, A-13
 - process flow (figure), A-2
 - process model location (table), A-3
 - selecting the default process model, A-6
 - Sequential, A-5, A-6
 - Configuration entry points, A-6
 - Engine callbacks, A-10
 - Execution entry points, A-6
 - Model callbacks, A-7
 - Single Pass entry point, A-12
 - Test UUTs entry point, A-11
 - utility sequences, A-10
 - station model, 1-14
 - support files, installation (table), A-33
- programming examples (NI resources), F-1
- project file, 2-5
- properties
 - built-in
 - sequence properties, 1-13
 - shared (note), 3-1
 - custom
 - lifetime, 3-3
 - result, 3-8
 - step type, 13-9
 - monitoring, 3-2
 - object reference properties, 12-5
 - property-array property, 1-8
 - Result.Status standard values (table), 3-16
 - shared built-in properties (note), 3-1
 - single-valued property, 1-8
 - using in expressions, 1-10

Property Browser panel, 4-5

Property Loader step, C-5

Q

Queue step, B-8

R

reading parameter information, 5-7

remote sequence execution, 5-17

Distributed COM (DCOM),
configuring, 5-18

Windows Vista, 5-18

Windows XP SP2, 5-18

security configuration

setting Windows system

security, 5-19

Windows 2000 SP4, 5-22

Windows Vista, 5-20

Windows XP SP2, 5-20

setting up TestStand as server, 5-19

security permissions, 5-19

Rendezvous step, B-7

report generation, 3-13

functions and sequences, A-36

header and footer (table), A-37

Model callbacks (table), A-38

report body (table), A-37

reports

Batch, 6-15

failure chain, 6-15

generating, on-the-fly, 6-16

implementing test resorts, 6-14

property flags, 6-15

result collection, 3-7

schema, XML, 6-16

using test reports, 6-14

ReportView control

description (table), 9-6

Requirements panel, 4-5

resolving type conflicts, 11-2

resource files

creating, 8-7

customizing, 8-7

escape codes (table), 8-8

format, 8-7

resource string files

escape codes (table), 8-8

Resource Usage Profiler, B-5

result collection, 1-16

custom result properties, 3-8

disabling, 3-7

loop results, 3-13

report generation, 3-13

standard result properties, 3-10

Results property, 12-6

root interactive execution, 3-5

route specification string, D-8

Run Options panel, 4-3

Run VI Asynchronously step, E-1

run-time errors, 1-12, 3-18

caution, 4-4

description, 3-18

handling, 3-18

S

schema

See also databases

specifying (tutorial), 6-12

search directories. *See* search paths

search paths, 5-2

deployment, configuring for, 5-3

Edit Search Directories dialog box, 5-2

Select step, 4-19

Semaphore step, B-17

Sequence Adapter, 1-7, 4-15, 5-16

Sequence Call step, 2-5, 4-15

sequence context, 3-2

Logging property, 6-7

using, 3-2

- sequence editor, 1-2
 - configuring startup options (table), 8-9
 - Execution window, 3-3
 - Workspace pane, 2-6
- sequence execution, 1-17
 - See also* execution
- sequence file global variables, 2-2
- sequence file translators, 15-1
 - creating translator DLLs, 15-2
 - examples, 15-2
 - using, 15-1
 - versioning, 15-3
- Sequence File window, 2-4
 - figure, 2-4
 - Steps pane, 2-5
 - Variables pane, 2-5
- sequence files, 1-1, 1-13, 2-1, 11-5
 - callbacks, 2-2
 - client sequence file, 1-15
 - comparing, 2-2
 - deploying custom sequence files, 15-4
 - editing
 - callback sequences, 10-2
 - entry point sequences, 10-3
 - normal sequences, 10-2
 - Front-End callbacks, 2-1
 - global variable, 1-13
 - global variables, 2-2
 - merging, 2-2
 - model, 2-1
 - normal, 2-1
 - processing with TestStand Deployment Utility, 14-4
 - special editing capabilities for process
 - model sequence files
 - callback sequences, 10-2
 - entry point sequences, 10-3
 - Station callbacks, 2-1
 - type concepts, 11-5
 - type definitions, 2-2
 - types of sequence files, 2-1
 - versioning custom sequence files, 15-3
 - views, 2-4
- Sequence Hierarchy window, 2-5
- sequence local variables, 1-12
- SequenceContext object, 1-9
- SequenceFileView Manager control, 9-4
- sequences, 1-1, 1-11, 2-3
 - built-in properties, 1-13
 - callback sequences, 1-16
 - callbacks, 1-17
 - entry point, 10-3
 - executing directly, 3-4
 - local variables, 1-12, 2-4
 - Model callbacks, 10-2
 - multithreading options, 1-17
 - parameters, 1-13, 2-3
 - run-time errors, 1-12
 - step groups, 1-12, 2-3
- Sequences pane, 2-5
- SequenceView control
 - description (table), 9-6
- Sequential process model, A-5, A-6
 - Configuration entry points, A-6
 - Engine callbacks, A-10
 - Execution entry point, A-6
 - Model callbacks, A-7
 - Single Pass entry point, A-12
 - Test UUTs entry point, A-11
 - utility sequences, A-10
- Session Manager, D-2
- shut down, 9-20
- Single Pass – Test Socket entry point
 - Batch process model, A-32
 - Parallel process model, A-19
- Single Pass Execution entry point, 10-3, A-4
 - Batch process model, A-31
 - Parallel process model, A-19
 - Sequential process model, A-12
- software (NI resources), F-1
- source code control (SCC), 2-6

- source code template, 5-2
 - See also* code templates
- SQL. *See* structured query language (SQL)
- standard named data types, 1-9, 12-5
 - CommonResults, 12-6
 - Error, 12-6
 - Path, 12-6
 - using, 12-5
- standard result property, 3-10
- startup, 9-20
- Statement step, 4-20
- station global variables, 1-8, 11-5
- Station Globals window, 11-5
- station model, 1-14
- StatusBar control
 - description (table), 9-7
- step groups, 2-3
- step properties
 - built-in, 4-3
 - Case, 4-19
 - lifetime of custom step properties, 3-3
 - modifying custom step properties, 4-5
 - Pass/Fail Test, 4-9
 - Select, 4-19
 - String Value Test, 4-14
- Step Properties dialog box
 - Code Templates tab, 13-6
 - Disable Properties tab, 13-5
 - General tab, 13-3
 - Menu tab, 13-3
 - Substeps tab, 13-4
 - Version tab, 13-8
- step results
 - See also* result collection
 - custom result properties (table), 3-8
 - exceptions, 3-10
 - loop results, 3-13
 - standard properties (table), 3-10
 - subsequences (table), 3-11
- Step Settings pane
 - Properties tab, 4-3
 - Additional Results panel, 4-5
 - Expressions panel, 4-5
 - General panel, 4-3
 - Looping panel, 4-4
 - Post Actions panel, 4-4
 - Preconditions panel, 4-5
 - Property Browser panel, 4-5
 - Requirements panel, 4-5
 - Run Options panel, 4-3
 - Switching panel, 4-4
 - Synchronization panel, 4-4
- step status, 3-16
 - failures, 3-17
 - terminations, 3-18
- step templates, 13-1
 - custom step types, differences with, 13-1
- step types
 - See also* built-in step types; Database step types; Flow Control step types; IVI step types; LabVIEW Utility step types; Synchronization step types
 - application-specific (note), 4-1
 - common properties, 13-2
 - custom, 13-1
 - creating, 13-1
 - step templates, differences with, 13-1
 - custom properties, 13-9
 - Insert Step submenu, 4-1
 - Insertion Palette, 4-1
 - module adapters, using with, 4-7
 - shared custom properties, 4-6
 - source code templates, 1-11
 - specifying multiple code templates, 13-8

- steps, 1-1, 1-10
 - built-in properties, 4-3
 - custom properties, 1-10
 - execution, order of actions (table), 3-14
 - failures, 3-17
 - interactive execution, 3-5
 - status (table), 3-16
 - step types, 1-11
 - terminations, 3-18
- Steps pane, 2-3, 2-5
- string resource files
 - creating, 8-7
 - customizing, 8-7
 - escape codes (table), 8-8
 - formatting, 8-7
- String Value Test step, 4-13
- structured query language (SQL)
 - SELECT command (queries), 6-2
 - SQL statement data, 6-2
 - step types. *See* Database step types
- submenus
 - Insert Step, 4-1
- subsequences, 1-1
 - results (table), 3-11
- substeps
 - custom, 13-5
 - Edit, 13-4
 - Post-Step, 13-4
 - Pre-Step, 13-4
- Switching panel, 4-4
- Synchronization object, B-1
 - common attributes
 - lifetime, B-4
 - name, B-3
 - timeout, B-4
 - resources, profiling usage of. *See* Resource Usage Profiler
- Synchronization panel, 4-4

- Synchronization step types, B-1
 - Auto Schedule, B-15
 - Batch Specification, B-19
 - Batch Synchronization
 - mismatched section, B-14
 - nested section, B-14
 - one thread only section, B-14
 - parallel section, B-14
 - serial section, B-13
 - step properties, B-15
 - synchronized section, B-13
 - Lock, B-5
 - Notification, B-10
 - Queue, B-8
 - Rendezvous, B-7
 - Semaphore, B-17
 - Thread Priority, B-17
 - Use Auto Scheduled Resource, B-16
 - Wait, B-11
- system deployment, 2-6

T

- tables
 - database result, 6-8
 - default result, creating, 6-9
 - query
 - SQL statement data, 6-2
- template
 - code, 13-6
 - location (table), 13-6
 - multiple, specifying, 13-8
 - legacy code, 13-7
 - source code, 5-2
 - step, 13-1
 - custom step types,
 - differences with, 13-1
- terminating executions, 3-6
- test executive, 1-1
- test modules. *See* code modules

- test reports
 - See also* reports; report generation
 - implementing, 6-14
 - using, 6-14
- Test step types. *See* built-in step types
- Test UUTs – Test Socket entry point
 - Batch process model, A-29
 - Parallel process model, A-18
- Test UUTs entry point
 - Batch process model, A-27
 - definition, 10-3, A-6
 - Parallel process model, A-17
 - Sequential process model, A-11
- tests, distributing, 14-7
- TestStand
 - architecture overview, 1-1
 - building blocks
 - automatic result collection, 1-16
 - callback sequences, 1-16
 - custom data types, 1-9
 - expressions, 1-10
 - process models, 1-14
 - properties, 1-8
 - sequence executions, 1-17
 - sequence files, 1-13
 - sequences, 1-11
 - standard data types, 1-9
 - steps, 1-10
 - variables and properties, 1-8
 - software components
 - module adapters, 1-7
 - sequence editor, 1-2
 - TestStand Engine, 1-6
 - TestStand User Interface (UI)
 - Controls, 1-6
 - user interfaces, 1-3
 - customizable components (table), 8-1
 - directories. *See* directory structure
 - general concepts, 1-1
 - search paths, 5-2
 - deployment, configuring for, 5-3
 - Edit Search Directories
 - dialog box, 5-2
 - software components, 1-2
- TestStand architecture overview
 - system components, 14-1
- TestStand Database Viewer, 6-9
 - result tables, creating (tutorial), 6-13
- TestStand Deployment Utility, 2-6, 14-1
 - deployment
 - building, 14-3
 - configuring, 14-3
 - identifying components for, 14-2
 - deployment scenarios, common
 - dynamically called files, adding to workspaces, 14-8
 - TestStand Engine, deploying, 14-6
 - user interfaces, distributing, 14-10
 - workspaces, distributing
 - tests from, 14-7
 - files, collecting, 14-3
 - guidelines, 14-5
 - installer, creating, 14-2
 - NI components, installing, 14-5
 - path references, 14-4
 - search paths, configuring, 5-3
 - sequence files, processing, 14-4
 - setup, 14-1
 - system components, 14-1
 - system workspace files, creating, 14-2
 - VIs, processing, 14-4
- TestStand Engine, 1-6
 - engine callbacks, 1-16
- TestStand process models. *See* Batch process model; Parallel process model; process models; Sequential process model

TestStand User Interface (UI)

Controls, 1-6, 9-2

See also connections; Manager controls;
visible controls

application styles, 9-26

multiple window, 9-27

no visible window, 9-29

single window, 9-27

caption connections, 9-10

command connections, 9-9

command-line arguments, 9-29

configuration file location, 9-30

connecting, 9-7

custom application settings, adding, 9-31

custom user interfaces,

documenting, 9-31

Editor applications, creating, 9-13

guidelines, 9-2

handling events, 9-17

DisplayExecution, 9-19

DisplaySequenceFile, 9-19

ExitApplication, 9-18

ReportError, 9-19

typical events, 9-18

Wait, 9-18

image connections, 9-11

interface pointer, obtaining, 9-17

license checking, 9-13

list connections, 9-8

localization, 9-25

Manager controls, 9-3

Application Manager, 9-3

ExecutionView Manager, 9-4

SequenceFileView Manager, 9-4

numeric value connections, 9-12

Operator Interface application, 9-13

persistence, 9-30

specifying and changing

connections, 9-12

startup and shutdown, 9-20

TSUtil Functions Library, 9-21

using with LabVIEW, 9-14

using with Visual C++, 9-16

using with Visual Studio, 9-15

view connections, 9-7

visible controls (table), 9-5

writing applications, 9-3

TestStand User Manager, 7-1

security (note), 7-1

TestStand Utility Functions Library (TSUtil)

assembly references in Visual Studio,

adding, 9-23

creating menu items, 9-23

localization functions, 9-26

updating menus, 9-24

Thread Priority step, B-17

Threads pane, 3-5

Tools menu, 8-2

training and certification (NI resources), F-1

translators, 15-1

deploying, 15-4

DLLs, creating, 15-2

examples, 15-2

using, 15-1

versioning, 15-3

troubleshooting (NI resources), F-1

type conflicts. *See* types

type definitions, 2-2

type palette files, 11-4

Type Properties dialog box, 12-7

types

See also built-in; data type; step type

conflicts, resolving, 11-2

creating, 11-1

modifying, 11-1

storing, 11-1

type palette files, 11-4

Types Window, 11-4

versioning, 11-2

Types pane, 2-2

Types window, 11-4

U

- unit under test (UUT), 1-2
- Use Auto Scheduled Resource step, B-16
- user interface controls *See* TestStand User Interface (UI) Controls
- user interfaces, 1-1, 1-3
 - application manifests, 9-33
 - application styles, 9-26
 - Authenticode signatures, 9-32
 - creating, 9-1
 - See also* connections; Manager controls; TestStand User Interface (UI) Controls; visible controls
 - deploying. *See* TestStand Deployment Utility
 - distributing, 14-10
 - documenting custom, 9-31
 - example user interfaces, 9-1
 - localization, 9-25
 - menu and menu items, 9-23
 - updating, 9-24
 - shutting down, 9-20
 - starting up, 9-20
 - startup options
 - configuring (table), 8-9
 - TestStand User Interface (UI) Controls, 1-6, 9-2
- user manager, 11-5
- User Manager window, 11-5
- User object, 7-3
- user privileges
 - accessing
 - current user, 7-2
 - verifying, 7-2
- using DLLs
 - MFC, 5-5
- utility subsequences
 - Batch process model, A-27
 - Parallel process model, A-13
 - Sequential process model, A-10

V

- variables
 - See also* properties
 - expressions, using, 1-10
 - local, 2-4
 - lifetime, 3-3
 - monitoring, 3-2
 - sequence file global, 2-2
 - sequence local, 1-12
 - standard and custom data types, 1-10
 - station global, 1-8
- Variables pane, 2-2, 2-3, 2-4, 2-5, 3-6, 12-4
- VariablesView control
 - description (table), 9-7
- VI processing
 - using the TestStand Deployment Utility, 14-4
- view connections, 9-7
- View Types For pane, 11-4
- visible controls
 - description of controls (table), 9-5
 - Manager controls, connecting, 9-7
- Visual Basic. *See* Microsoft
- Visual C++. *See* Microsoft
- Visual Studio .NET 2003. *See* Microsoft
- Visual Studio 2005. *See* Microsoft
- Visual Studio. *See* Microsoft

W

- Wait step, B-11
- Watch View pane, 3-6
- While step, 4-18
- windows
 - Execution, 3-3
 - Sequence File, 2-4
 - Sequence Hierarchy, 2-5
 - Station Global, 11-5
 - Types, 11-4
 - User Manager, 11-5

- Windows 2000 SP4
 - remote execution
 - security, setting, 5-22
- Windows remote execution
 - 2000 Service Pack 4, 5-22
 - XP Service Pack 2, 5-20
- Windows Vista
 - application manifests, 9-33
 - Authenticode signatures, 9-32
 - Distributed COM (DCOM),
 - configuring, 5-18
 - remote execution
 - firewall settings, configuring, 5-21
 - security, setting, 5-20
 - TestStand, using with
 - directory structure, 8-2
- Windows XP SP2
 - Distributed COM (DCOM),
 - configuring, 5-18
 - remote execution
 - firewall settings, configuring, 5-21
 - security, setting, 5-20
- workspace files
 - adding dynamically called files, 14-8
 - creating, 14-2
 - source code control, 2-6
 - tests, distributing, 14-7
- Workspace pane, 2-6
- workspaces, 2-5

X

- XML report schema, 6-16