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, LOSS OF PRODUCTS, OR INJURY OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. THIS LIMITATION OF 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 REASONABLE CONTROL. THE WARRANTY PROVIDED HEREIN DOES NOT APPLY TO 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 NEGLECT; 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 is 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
CVI, LabVIEW, National Instruments, NI, ni.com, NI TestStand, the National Instruments corporate logo, and the Eagle logo are trademarks of National Instruments Corporation. Refer to the Trademark Information at ni.com/trademarks for other National Instruments trademarks.

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

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

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

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 HERE AFTER COLLECTIVELY TERMED “SYSTEM FAILURES”). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS’ TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.
Conventions

The following conventions are used in this manual:

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

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

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

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 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.

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.

Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

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

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 ................................................................. 1-4
    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 Deployment Utility ......................................................... 1-8

TestStand Building Blocks ..................................................................... 1-8
Properties ............................................................................................. 1-8
  Built-In and Custom Properties ......................................................... 1-9
Variables ............................................................................................... 1-9
Standard and Custom Data Types ......................................................... 1-9
Expressions .......................................................................................... 1-10
Steps ........................................................................................................ 1-11
  Step Types .......................................................................................... 1-11
Sequences .............................................................................................. 1-12
  Step Groups ....................................................................................... 1-12
  Sequence Local Variables ................................................................. 1-13
  Sequence Parameters ......................................................................... 1-13
    Built-in Sequence Properties ......................................................... 1-14
Sequence Files ....................................................................................... 1-14
Process Models ...................................................................................... 1-14
  Specifying Process Model Files ......................................................... 1-15
  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 ...................................................................................... 2-1
- Sequence File Globals ......................................................................................... 2-2
- Sequence File Type Definitions ....................................................................... 2-2
- Comparing and Merging Sequence Files ....................................................... 2-2
- Password Protecting Sequence Files ................................................................. 2-2
- Analyzing Sequence Files ................................................................................... 2-2
- Sequences ...................................................................................................................... 2-3
- Step Groups ......................................................................................................... 2-3
- Parameters ........................................................................................................... 2-3
- Local Variables .................................................................................................... 2-3
- 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 .............................................................. 3-7
  - Checking for Suspended or Stopped Execution within Code Modules ............ 3-7
- Result Collection ....................................................................................................... 3-8
  - Custom Result Properties .............................................................................. 3-9
    - Exceptions ................................................................................................ 3-9
  - Standard Result Properties ............................................................................ 3-11
  - Subsequence Results ....................................................................................... 3-12
  - Loop Results ................................................................................................... 3-14
  - Report Generation ........................................................................................... 3-14
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 ................................................................. 5-3
LabWindows/CVI Adapter ....................................................... 5-4
C/C++ DLL Adapter ................................................................. 5-4
  Accessing the TestStand API in Visual Studio ....................... 5-5
  TestStand Include Directories ............................................ 5-5
Using DLLs ............................................................................... 5-6
  Using ActiveX Controls in LabVIEW DLLs ......................... 5-6
  Using MFC in DLLs .............................................................. 5-6
  Loading Subordinate DLLs .................................................. 5-6
  Reading Parameter Information ......................................... 5-7
Debugging DLLs ....................................................................... 5-8
  Debugging LabVIEW Shared Libraries (DLLs) .................... 5-9
.NET Adapter ................................................................. 5-10
  Generics ........................................................................ 5-11
  Debugging .NET Assemblies .............................................. 5-11
  Using the .NET Framework ............................................. 5-12
  Accessing the TestStand API in Visual Studio ................. 5-13
ActiveX/COM Adapter .......................................................... 5-14
  Debugging ActiveX Automation Servers ......................... 5-14
  Registering and Unregistering ActiveX/COM Servers .......... 5-15
HTBasic Adapter ................................................................. 5-15
  Debugging HTBasic Subroutines ...................................... 5-15
Sequence Adapter ............................................................... 5-16
  Remote Sequence Execution ......................................... 5-16
  Setting up TestStand as a Server
    for Remote Sequence Execution ..................................... 5-18
  Setting Windows System Security on a Remote Server ....... 5-19
  Setting Windows System Security on a Local Client .......... 5-25
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
Report Generation Implementation ....................................... 6-14
Using Reports ................................................................. 6-15
Using Expressions to Customize Reports ............................. 6-15
Failure Chain in Reports ..................................................... 6-17
Batch Reports ................................................................. 6-17
Property Flags that Affect Reports ....................................... 6-17
On-the-Fly Report Generation ............................................... 6-17
XML Reports ................................................................. 6-18
Transforming XML Reports ................................................ 6-19
Associating Style Sheets with XML Reports ......................... 6-19
Example HTML Transformation ........................................ 6-20
Distributing XML Reports .................................................. 6-21
Displaying Measurement Data ............................................ 6-21

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 .................................................................................................. 8-3
    Components Directory ................................................................................................ 8-4
  <TestStand Public> Directory ....................................................................................... 8-5
    Copying Read-Only Files to Modify .......................................................................... 8-6
  <TestStand Application Data> Directory ....................................................................... 8-7
  <TestStand Local Application Data> Directory ............................................................ 8-8
Creating String Resource Files ........................................................................................ 8-9
  String Resource File Format ....................................................................................... 8-10
Configuring Sequence Editor and User Interface Startup Options ................................. 8-11
Configure Menu .............................................................................................................. 8-13

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
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
Managing Type Revisions ............................................................................ 11-3
Avoiding Unwanted Type Version Propagation ........................................... 11-4
Avoiding Unwanted Type Version Propagation to Previous Versions of TestStand .......... 11-5
Using Type Palette Files to Restrict Automatic Type Conflict Resolution ... 11-5
Types Window ................................................................................................ 11-7
Type Palette Files ....................................................................................... 11-8
Sequence Files ........................................................................................... 11-8
Station Globals ......................................................................................... 11-9
User Manager ............................................................................................ 11-9

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 ............................................................................................................. 12-7
Expression .................................................................................................. 12-7
Creating Custom Data Types ........................................................................ 12-7
Properties Common to All Data Types ........................................................ 12-8
Custom Properties of Data Types ................................................................. 12-9

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 .................................................................................................... 13-4
Substeps Tab ............................................................................................... 13-4
Disable Properties Tab ............................................................................... 13-6
Chapter 14
Deploying TestStand Systems
TestStand System Components .......................................................... 14-1
Creating a Deployment ................................................................. 14-2
  Identifying Components to Deploy ................................................ 14-2
  Organizing Files for Deployment .................................................. 14-3
Configuring the Deployment ............................................................ 14-3
Determining Whether to Create an Installer ..................................... 14-3
Deployment Utility Build Process ....................................................... 14-4
  Collecting Files ........................................................................... 14-4
    Filtering Files ........................................................................... 14-4
  Processing VIs .......................................................................... 14-6
  Processing Sequence Files ......................................................... 14-7
  Installing National Instruments Components .................................. 14-7
Guidelines for Successful Deployment ............................................. 14-8
Common Deployment Scenarios ....................................................... 14-9
  Deploying the TestStand Engine .................................................. 14-10
  Distributing Tests from a Workspace ............................................ 14-11
  Adding Dynamically Called Files to a Workspace ........................ 14-12
  Distributing a User Interface ...................................................... 14-14

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
Contents

Appendix C  
Database Step Types

Appendix D  
IVI Step Types

Appendix E  
LabVIEW Utility Step Types

Appendix F  
Technical Support and Professional Services

Index
NI TestStand Architecture

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

National Instruments recommends that you read the \textit{NI TestStand System and Architecture Overview Card} and the \textit{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.
Chapter 1  NI TestStand Architecture

- **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.

- **Deployment**—An image or an installer for an image to transfer a test system from development to production.

**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 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
- Performing static analysis of sequence files to locate errors and enforce coding guidelines
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 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 is an application you deploy to a development system or a production station to provide a custom GUI for executing, debugging, or editing sequences. Simple user interfaces might only support running sequences, and custom sequence editors might support editing, running, and debugging sequences.

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>
<thead>
<tr>
<th>Features</th>
<th>TestStand Sequence Editor</th>
<th>User Interface Editor Mode</th>
<th>User Interface Operator Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docking, hiding, and floating panes</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Configurable menus and toolbars</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Navigation among sequences</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Sequence Hierarchy window</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
Table 1-1. Features of TestStand Sequence Editor and TestStand User Interfaces (Continued)

<table>
<thead>
<tr>
<th>Features</th>
<th>TestStand Sequence Editor</th>
<th>User Interface Editor Mode</th>
<th>User Interface Operator Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrated sequence analysis</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>User management configuration</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>User privileges enforced</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Configurable step list</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Workspace support</td>
<td>Dockable pane</td>
<td>Modal dialog</td>
<td>Modal dialog</td>
</tr>
<tr>
<td>Source code control support</td>
<td>Dockable pane</td>
<td>Modal dialog</td>
<td>—</td>
</tr>
<tr>
<td>Configure report generation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Configure database logging</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Configure station options</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Editing**

<table>
<thead>
<tr>
<th>Features</th>
<th>TestStand Sequence Editor</th>
<th>User Interface Editor Mode</th>
<th>User Interface Operator Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit sequence files</td>
<td>✓</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>Insertion Palette</td>
<td>✓</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>Edit steps and modules</td>
<td>Dockable pane</td>
<td>Modal dialogs</td>
<td>—</td>
</tr>
<tr>
<td>Integration with ADEs</td>
<td>Dockable pane</td>
<td>Modal dialogs</td>
<td>—</td>
</tr>
<tr>
<td>Edit variables and station globals</td>
<td>✓</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>Edit types</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Edit process models</td>
<td>✓</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>Multiple step editing</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Undo and redo</td>
<td>✓</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>Find and replace</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Integrated file differ</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**Running**

<table>
<thead>
<tr>
<th>Features</th>
<th>TestStand Sequence Editor</th>
<th>User Interface Editor Mode</th>
<th>User Interface Operator Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multithreaded execution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Single-step debugging</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Conditional breakpoints</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Call stack and thread lists</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
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 UI Controls use the TestStand API, which you can call from any programming environment that supports access to ActiveX servers, including code modules you write in LabVIEW and LabWindows/CVI.

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

<table>
<thead>
<tr>
<th>Features</th>
<th>TestStand Sequence Editor</th>
<th>User Interface Editor Mode</th>
<th>User Interface Operator Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables view</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Watch view</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Output messages view</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Can include in deployment</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Source code available</td>
<td>—</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Minimum license required</td>
<td>TestStand Development System License</td>
<td>TestStand Custom Sequence Editor License</td>
<td>TestStand Base Deployment Engine License</td>
</tr>
</tbody>
</table>
Module Adapters

The TestStand Engine uses module adapters to invoke code modules that 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. The VIs can be in LabVIEW packed project libraries, in projects, on disk, or in LLBs. Refer to Chapter 7, *Effectively Using LabVIEW with TestStand*, of the *Using LabVIEW and LabWindows/CVI with TestStand* manual for more information about how to best use LabVIEW features in a TestStand system.

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

- **LabWindows/CVI Adapter**—Calls C functions with a variety of parameter types. The functions can be in object files, library files, or DLLs. They can also be source files located in the project you are currently using in LabWindows/CVI.

- **C/C++ DLL Adapter**—Calls functions or methods in a DLL with a variety of parameter types, including supported versions of 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 or COM 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 Deployment Utility

After you develop, customize, and debug a TestStand system, you can deploy the system to multiple test stations. 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, configuration data for instruments, and support files for the test system and creating an installer for the files. Refer to Chapter 14, *Deploying TestStand Systems*, for more information about the deployment utility.

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**—A single value. TestStand supports number, string, Boolean, and object reference single-valued properties.
- **Array property**—An array of values. TestStand supports number, string, Boolean, and object reference array properties.
- **Property-array property**—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. When 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 and the expression operators and syntax TestStand supports.

**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 causes a run-time error to occur or if the operator terminates the execution, the flow of execution stops and jumps to the Cleanup step group. The Cleanup steps always run even when some of the Setup steps do not run. When a Cleanup step causes a run-time error, execution continues to the next Cleanup step.

When 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 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 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 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 file and to specify whether individual sequence files can use their own process model files. When 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 that 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 that invokes a test sequence 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.
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.

Table 1-2. Callback Types

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

Sequence files and workspaces help you organize your work.

Sequence Files

A 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 normal sequence files, process model sequence files, Station callback sequence files, and Front-End callback sequence files.

Most sequence files you create are normal sequence files that include sequences to test UUTs. Usually, an application has one Station callback sequence file and one Front-End callback sequence file.

Normal sequence files specify that 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 on 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 on 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 TestStand File Diff/Merge application is a stand-alone application you can use to compare and merge non-type differences between two or three sequence files and compare type differences among two sequence files or type palette files.

Refer to the NI TestStand Help for more information about the file diff/merge application.

Password Protecting Sequence Files

TestStand supports password protecting sequence files to discourage editing and viewing within the sequence editor and user interfaces that use the TestStand User Interface (UI) Controls. The TestStand API limits access to a file protected from viewing but cannot prevent access to the file content during execution. National Instruments does not recommend using passwords as the only way of protecting intellectual property.

Analyzing Sequence Files

Use the TestStand Sequence Analyzer in the TestStand Sequence Editor or the stand-alone sequence analyzer application to find errors, enforce custom development guidelines you establish, and gather statistics about workspace files, sequence files, directories, type palette files, station globals files, template files, and users files during development or before deployment. The sequence analyzer uses a built-in set of rules and analysis modules to analyze the files you specify and generate messages that
correspond to each issue found during analysis. The built-in rules are
designed to detect, at edit time, the most common situations that can cause
run-time failures.

Refer to the *NI TestStand Help* for more information about the sequence
analyzer, rules, and analysis modules.

## 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 a 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](image)

To open an existing sequence file in the Sequence File window, select File » Open 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.
Chapter 2  Sequence Files and Workspaces

- **Steps**—Displays the steps in a specific sequence. Expand the Setup, Main, or Cleanup group to view the contents of the group.

- **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 panes.

### Sequence Hierarchy Window

TestStand sequences can use the Sequence Call step type to call subsequences. Use the Sequence Hierarchy window in the sequence editor to display a graph of the sequence call and callback relationships among sequence files to more easily visualize, navigate, and maintain test sequences. 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 workspace file (.tsw) contains references to any number of TestStand project files. A project file (.tpj) contains references to any number of other files of any type.

Use 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 File**. To create a new workspace file, select **File»New»Workspace File**.
Note When you modify or replace a file in a workspace, TestStand reflects the changes only when the file maintains the same filename and 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 search directories.

TestStand creates an options file (.tso) that has the same name as the workspace file and is located in the same directory as the workspace file. This configuration file contains breakpoints you set when the workspace is loaded, watch expressions, and the expansion state of the Workspace pane. 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 can use a workspace and project files to specify all the required files the TestStand Deployment Utility collects 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 the 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:

```plaintext
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 TestStand 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.

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 unit under test (UUT) identification and report generation. For example, the default 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 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 whether 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. When you used the process model for the original execution, the report includes these results. When the selected steps complete, the execution returns to the originally suspended step. A configurable station option specifies whether 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 on the Execution tab of the Station Options dialog box. You can also use the Branching Mode control in this dialog box to configure whether 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 commands in the Debug menu to suspend executions, control how TestStand single-steps within executions, set breakpoints, and stop executions prematurely. Executions suspend only after completing the execution of the current step in all threads of an execution.
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. TestStand updates these panes while tracing. 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 when executions suspend. 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. When you enable 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.

  **Caution**  Watch View pane expressions that modify the structure of objects, such as deleting subproperties or changing the size of an array, can prevent the Variables pane from refreshing correctly or can cause the Variables pane to generate errors.

- **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 whether 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 for each thread in the execution returns control.

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

When you terminate an execution, all the Cleanup steps, including subsequences the Cleanup steps call, execute normally and disregard the pending termination. If you terminate an execution, or reselect Terminate while a Cleanup step is running and the step monitors for termination, the step terminates prematurely and TestStand proceeds to execute the next Cleanup step. Terminating a Cleanup step terminates only that step and the execution proceeds with the next Cleanup step. When you terminate an execution while the client sequence file is still running, the default process model continues to run, possibly testing the next UUT or generating a 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.

Checking for Suspended or Stopped Execution within Code Modules

Code modules that TestStand calls can launch dialog boxes or perform other time-consuming operations. In these cases, it can be useful for code modules to periodically check whether TestStand terminated or aborted the parent execution of the code module so the code module can stop gracefully and allow the parent execution to terminate or abort. For example, you might want a code module to dismiss a dialog box or cancel a long acquisition and immediately return to TestStand when execution of the code module terminates.

In addition, it can be useful for code modules to permit TestStand to suspend the execution without requiring the code module to first return to TestStand. For example, you might want TestStand to be able to suspend an execution at a breakpoint while a code module displays a dialog box that waits for user input.
Use the `Execution.InitTerminationMonitor` and `Execution.GetTerminationMonitorStatus` methods to determine whether the execution receives a request to terminate or abort the execution. The monitor only recognizes requests to terminate or abort while monitoring, so a code module that executes in a Cleanup step group of an already terminating execution monitors for a subsequent request to terminate the step or abort the execution.

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

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

Refer to the TestStand VIs for LabVIEW and to the TestStand Utility Functions for LabWindows/CVI for utility functions that check for suspended or stopped execution. Refer to the Other User Interface Utilities sections of Chapter 6, Creating Customer User Interfaces in LabVIEW, and Chapter 15, Creating Customer User Interfaces in LabWindows/CVI, of the Using LabVIEW and LabWindows/CVI with TestStand manual for more information about utility functions in LabVIEW and LabWindows/CVI, respectively.

Refer to Chapter 7, Effectively Using LabVIEW with TestStand, of the Using LabVIEW and LabWindows/CVI with TestStand manual for more information about how to best use LabVIEW features in a TestStand system.

**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 the Result subproperty of the step, 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, the position of the step 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 defines a set of custom properties and can have a different set of subproperties under the Result property, the step result varies according to the step type. 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 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.
  - When this property is False, TestStand reports the run-time error to the sequence.
When this property is True, TestStand continues execution normally after the step.

- **Step.Result.Status**—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**—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.

**Note** The preceding list does not include the result properties that are unique for each built-in step type or for Synchronization, Database, Interchangeable Virtual Instrument (IVI), or LabVIEW Utility step types. Refer to Chapter 4, *Built-In Step Types*, for more information about custom properties of the built-in step types and 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 those 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. When you modify the CommonResults data type, you must ensure that the new type definition does not conflict with earlier versions of the type in other files. Refer to Chapter 11, *Type Concepts*, for more information about modifying and versioning types. National Instruments recommends modifying the CommonResults data type only when 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 3-1.

<table>
<thead>
<tr>
<th>Standard Result Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS.StartTime</td>
<td>Time at which the step began executing; specifically, the number of seconds since the TestStand Engine initialized.</td>
</tr>
<tr>
<td>TS.TotalTime</td>
<td>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.</td>
</tr>
<tr>
<td>TS.ModuleTime</td>
<td>Number of seconds the code module took to execute.</td>
</tr>
<tr>
<td>TS.Index</td>
<td>Zero-based position of the step in the step group.</td>
</tr>
<tr>
<td>TS.StepName</td>
<td>Name of the step.</td>
</tr>
<tr>
<td>TS.StepGroup</td>
<td>Step group that contains the step. The values are Main, Setup, or Cleanup.</td>
</tr>
</tbody>
</table>
Table 3-1. Standard Step Result Properties (Continued)

<table>
<thead>
<tr>
<th>Standard Result Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS.StepId</td>
<td>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.</td>
</tr>
<tr>
<td>TS.Id</td>
<td>A number TestStand assigns to the step result. The number is unique with respect to all other step results in the current TestStand session.</td>
</tr>
<tr>
<td>TS.InteractiveExeNum</td>
<td>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 when you run the step interactively.</td>
</tr>
<tr>
<td>TS.StepType</td>
<td>Name of the step type.</td>
</tr>
<tr>
<td>TS.Server</td>
<td>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.</td>
</tr>
<tr>
<td>TS.StepCausedSequenceFailure</td>
<td>TestStand adds this property only when the step fails. The value is True when the step failure causes the sequence to fail. The value is False when the step failure does not cause the sequence to fail or when the sequence has already failed.</td>
</tr>
<tr>
<td>TS.BlockLevel</td>
<td>Indicates the number of blocks that encloses the step, such as If and For steps. The value is zero for top-level steps.</td>
</tr>
</tbody>
</table>

**Subsequence Results**

When 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 3-2 lists the name of the subproperty for each type of subsequence call.
### Table 3-2. Subproperty Names for Subsequence Results

<table>
<thead>
<tr>
<th>Result Subproperty Name</th>
<th>Type of Subsequence Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS.SequenceCall</td>
<td>Sequence Call</td>
</tr>
<tr>
<td>TS.PostAction</td>
<td>Post Action callback</td>
</tr>
<tr>
<td>TS.SequenceFilePreStep</td>
<td>SequenceFilePreStep callback</td>
</tr>
<tr>
<td>TS.SequenceFilePostStep</td>
<td>SequenceFilePostStep callback</td>
</tr>
<tr>
<td>TS.ProcessModelPreStep</td>
<td>ProcessModelPreStep callback</td>
</tr>
<tr>
<td>TS.ProcessModelPostStep</td>
<td>ProcessModelPostStep callback</td>
</tr>
<tr>
<td>TS.StationPreStep</td>
<td>StationPreStep callback</td>
</tr>
<tr>
<td>TS.StationPostStep</td>
<td>StationPostStep callback</td>
</tr>
<tr>
<td>TS.SequenceFilePreInteractive</td>
<td>SequenceFilePreInteractive callback</td>
</tr>
<tr>
<td>TS.SequenceFilePostInteractive</td>
<td>SequenceFilePostInteractive callback</td>
</tr>
<tr>
<td>TS.ProcessModelPreInteractive</td>
<td>ProcessModelPreInteractive callback</td>
</tr>
<tr>
<td>TS.ProcessModelPostInteractive</td>
<td>ProcessModelPostInteractive callback</td>
</tr>
<tr>
<td>TS.StationPreInteractive</td>
<td>StationPreInteractive callback</td>
</tr>
<tr>
<td>TS.StationPostInteractive</td>
<td>StationPostInteractive callback</td>
</tr>
<tr>
<td>TS.SequenceFilePostResultListEntry</td>
<td>SequenceFilePostResultListEntry callback</td>
</tr>
<tr>
<td>TS.ProcessModelPostResultListEntry</td>
<td>ProcessModelPostResultListEntry callback</td>
</tr>
<tr>
<td>TS.StationPostResultListEntry</td>
<td>StationPostResultListEntry callback</td>
</tr>
<tr>
<td>TS.SequenceFilePostStepRuntimeError</td>
<td>SequenceFilePostStepRuntimeError callback</td>
</tr>
<tr>
<td>TS.ProcessModelPostStepRuntimeError</td>
<td>ProcessModelPostStepRuntimeError callback</td>
</tr>
<tr>
<td>TS.StationPostStepRuntimeError</td>
<td>StationPostStepRuntimeError callback</td>
</tr>
<tr>
<td>TS.SequenceFilePostStepFailure</td>
<td>SequenceFilePostStepFailure callback</td>
</tr>
<tr>
<td>TS.ProcessModelPostStepFailure</td>
<td>ProcessModelPostStepFailure callback</td>
</tr>
<tr>
<td>TS.StationPostStepFailure</td>
<td>StationFilePostFailure callback</td>
</tr>
</tbody>
</table>
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 report by traversing the results for the Main sequence in the client sequence file and all the subsequences it calls.

### 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 3-3 lists the most common actions a step can take, in the order the step performs them.

<table>
<thead>
<tr>
<th>Action Number</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Allocate step result</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>Enter batch synchronization section</td>
<td>When option is set</td>
</tr>
<tr>
<td>3</td>
<td>Check run mode for Skip</td>
<td>Sets the corresponding status on the step, then proceeds to Action Number 31</td>
</tr>
<tr>
<td>4</td>
<td>Evaluate precondition</td>
<td>When False, performs Action Number 26, then proceeds to Action Number 30</td>
</tr>
<tr>
<td>5</td>
<td>Acquire step lock</td>
<td>When option is set</td>
</tr>
<tr>
<td>6</td>
<td>Check run mode for Force Pass or Force Fail</td>
<td>Sets the corresponding status on the step, then proceeds to Action Number 27</td>
</tr>
<tr>
<td>7</td>
<td>Load module if not already loaded</td>
<td>—</td>
</tr>
<tr>
<td>8</td>
<td>Execute step switching</td>
<td>—</td>
</tr>
</tbody>
</table>
Table 3-3. Order of Actions a Step Performs (Continued)

<table>
<thead>
<tr>
<th>Action Number</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Evaluate Loop Initialization expression</td>
<td>Only when looping</td>
</tr>
<tr>
<td>10</td>
<td>Evaluate Loop While expression, skip to Action Number 24 when False</td>
<td>Only when looping</td>
</tr>
<tr>
<td>11</td>
<td>Allocate loop iteration result</td>
<td>Only when looping</td>
</tr>
<tr>
<td>12</td>
<td>Call Pre-Step Engine callbacks</td>
<td>—</td>
</tr>
<tr>
<td>13</td>
<td>Evaluate Pre-Expression</td>
<td>—</td>
</tr>
<tr>
<td>14</td>
<td>Call Pre-Step substeps for step type</td>
<td>—</td>
</tr>
<tr>
<td>15</td>
<td>Call module</td>
<td>—</td>
</tr>
<tr>
<td>16</td>
<td>Call Post-Step substeps for step type</td>
<td>TestStand calls Post-Step substeps even when the user code module generates a run-time error, which enables Post-Step substeps to perform error handling, when appropriate. If Post-Step substeps clear run-time errors, proceed to Action Number 17. If step is configured to ignore run-time errors, proceed to Action Number 21.</td>
</tr>
<tr>
<td>17</td>
<td>Evaluate Post-Expression</td>
<td>—</td>
</tr>
<tr>
<td>18</td>
<td>Evaluate Status expression</td>
<td>—</td>
</tr>
<tr>
<td>19</td>
<td>Call Post-Step Engine callbacks</td>
<td>—</td>
</tr>
<tr>
<td>20</td>
<td>Call Post-Step Failure Engine callback</td>
<td>Only when loop iteration fails</td>
</tr>
<tr>
<td>21</td>
<td>Populate loop iteration result</td>
<td>Only when looping</td>
</tr>
<tr>
<td>22</td>
<td>Call Post-ResultList Entry Engine callback</td>
<td>Only when looping</td>
</tr>
<tr>
<td>23</td>
<td>Evaluate Loop Increment expression, return to Action Number 10</td>
<td>Only when looping</td>
</tr>
<tr>
<td>24</td>
<td>Evaluate Loop Status expression</td>
<td>Only when looping</td>
</tr>
<tr>
<td>25</td>
<td>Disconnect switching routes with step lifetime</td>
<td>—</td>
</tr>
</tbody>
</table>
Table 3-3. Order of Actions a Step Performs (Continued)

<table>
<thead>
<tr>
<th>Action Number</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>Unload module when required</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Update sequence failed state</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Call Post-Step Failure Engine callback</td>
<td>Only when step fails</td>
</tr>
<tr>
<td>29</td>
<td>Execute post action</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Release step lock</td>
<td>When option is set</td>
</tr>
<tr>
<td>31</td>
<td>Exit batch synchronization section</td>
<td>When option is set</td>
</tr>
<tr>
<td>32</td>
<td>Populate step result</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Call Post-ResultList Entry Engine callback</td>
<td></td>
</tr>
</tbody>
</table>

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 in one of these actions, it calls the Post-Step Error Engine callbacks. When you do not define these callbacks or when the callbacks do not reset the error state for the step, TestStand executes only the following actions to clean up the execution of the step:

- Disconnect switching routes with step lifetime
- Unload module when required
- Release step lock if set
- Exit batch synchronization section if set
- Populate step result
- Call Post-ResultList Entry Engine callback

TestStand skips all other actions, such as evaluating the post-expression and status expression. When a run-time error occurs in a loop iteration, TestStand also populates the loop iteration result.
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 the code module of the step can set the status property, TestStand and the built-in step types use and recognize the values that appear in Table 3-4.

<table>
<thead>
<tr>
<th>String Value</th>
<th>Meaning</th>
<th>Source of the Status Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passed</td>
<td>Indicates the step performed a test that passed, or the step did not execute because the run mode for the step is Force Pass.</td>
<td>Step, code module, or TestStand</td>
</tr>
<tr>
<td>Failed</td>
<td>Indicates the step performed a test that failed, or the step did not execute because the run mode for the step is Force Fail.</td>
<td>Step, code module, or TestStand</td>
</tr>
<tr>
<td>Error</td>
<td>Indicates a run-time error occurred.</td>
<td>TestStand</td>
</tr>
<tr>
<td>Done</td>
<td>Indicates the step completed without setting the status.</td>
<td>TestStand</td>
</tr>
<tr>
<td>Terminated</td>
<td>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 Terminated to a calling sequence, TestStand sets the step status to Terminated. 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 Terminated only if TestStand returns the request to terminate the execution to the root sequence invocation.</td>
<td>TestStand</td>
</tr>
<tr>
<td>Skipped</td>
<td>Indicates the step did not execute because the run mode for the step is Skip.</td>
<td>TestStand</td>
</tr>
<tr>
<td>Running</td>
<td>Indicates the step is currently running.</td>
<td>TestStand</td>
</tr>
<tr>
<td>Looping</td>
<td>Indicates the step is currently running in loop mode.</td>
<td>TestStand</td>
</tr>
</tbody>
</table>
Failures

When 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 waits for the current running step to complete before initiating termination. A terminating sequence runs steps in the Cleanup step group, if not already executing, and returns to the calling sequence. 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. When you enable 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 when it encounters a condition that prevents a sequence from executing. For example, when 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 when it detects an error in a hardware or software resource it uses to perform a test.

To interactively handle run-time errors, 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.

You can also 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 on an edit tab or in an 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.
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 to 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, click 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**—The name of the step.
- **Type**—The type of the step.
- **Adapter**—The adapter the step uses to call a code module.
- **Icon**—The icon to display for the step.
- **Comment**—The comment of the step.

#### Run Options Panel
- **Load/Unload Options**—Specifies how to load and unload the code modules or subsequences each step invokes.
- **Run Mode**—Specifies to skip a step, force the step to pass or fail without executing the code module of the step, or execute normally.
- **Precondition Evaluation in Interactive Mode**—Specifies whether to evaluate the step precondition when you run the step interactively.
- **TestStand Window Activation**—Specifies whether the TestStand application activates its window when the step completes.
- **Sequence Call Trace Setting**—Specifies whether to trace into the steps in the subsequence the step calls. This property exists only for Sequence Call steps.
- **Record Result**—Specifies whether to collect 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 whether to set the status of the sequence to Failed when the status of the step is Failed.
• **Ignore Run-Time Errors**—Specifies whether to continue 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 whether a Sequence Call step ignores the request to terminate execution.

### Looping Panel

• **Loop**—Specifies whether 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 the next TestStand operation depending on the pass/fail status of the step or any custom condition. Operations include executing the next step, executing a particular step or sequence, terminating the execution, or suspending in debug mode.

### Switching Panel

• **Switching**—Specifies whether TestStand performs switching operations when the step executes.

### Synchronization Panel

• **Synchronization**—Specifies whether a step blocks another instance of the step from executing at the same time in a different thread.

### Expressions Panel

• **Pre-Expression**—An expression to evaluate before executing the code module of the step.

• **Post-Expression**—An expression to evaluate after executing the code module of the step.

• **Status Expression**—An expression that determines the value of the status property of the step. TestStand evaluates this expression after evaluating the Post-Expression.
Preconditions Panel
Specifies the conditions that must evaluate to `True` for TestStand to execute a step during the normal flow of execution in a sequence, such as running a step only if a previous step passes.

Requirements Panel
Notates product and unit requirements a step covers.

Additional Results Panel
Adds and configures 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. You can configure TestStand to automatically include additional results in reports and when logging results to a database. The default TestStand report generation sequences and XML style sheets do not display additional results for skipped steps.

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.

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 TestStand 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>

**Note** In most cases, using the Sequence Call step to call subsequences is adequate. A Sequence Call step always uses the Sequence Adapter. By default, the Sequence Adapter is hidden in the Adapter ring control. Select **Configure»Adapters** from the menu bar and remove the checkmark from the checkbox in the Hidden column to display the Sequence Adapter in the Adapter ring control.

When 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. When `Step.Result.PassFail` is True, the step type sets the step status to Passed. When `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 and 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 whether the step passes or fails. To customize the Boolean expression that determines whether 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**—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**—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**—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 when 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

\[
\text{RunState.PreviousStep.Result.Numeric * Locals.Attenuation > 12}
\]

### 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. When 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 and 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**—The numeric measurement value. Usually, you set this value in the code module.
- **Step.Result.Units**—A label that indicates the unit of measurement.
- **Step.Limits.Low, High, LowExpr, HighExpr, UseLowExpr, and UseHighExpr**—The limits for the comparison.
- **Step.Comp**—The type of comparison, such as `<EQ`.
- **Step.CompExpr**—The comparison operation using an expression.
- **Step.UseCompExpr**—The step uses the expression to compare the measurement values.
- **Step.InBuf**—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**—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 when 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.
Note  The type for measurement and limit values must be double-precision, 64-bit floating-point values, and cannot be signed or unsigned 64-bit integers.

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 when all measurements of the step 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**—The limits for the comparison.
  - **Units**—A label that describes the measurement units for the limits and the measurement value.
  - **Comp**—The type of comparison, such as EQ.
  - **CompExpr**—The comparison operation using an expression.
  - **UseCompExpr**—The step uses 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**—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**—A numeric array that is the default data source `Step.DataSource` specifies.

• **Step.UseIndividualDataSources**—When this property is True, the step stores separate data source expressions for each measurement in the `Step.DataSourceArray`. When this property is False, the step obtains the data values for each measurement from the numeric array the `Step.DataSource` property specifies.

• **Step.DataSourceArray**—A data source for each measurement element in the measurement array.

• **Step.ExpectedNumMeasure**—The number of measurements for the step.

• **Step.ExtraDataAction**—Specifies how the step processes data when 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**—A measurement to repeat when the `Step.ExtraDataAction` is set to `RepeatOne`.

• **Step.ExtraMeasAction**—Specifies how the step responds when the numeric array contains fewer elements than the expected number of measurements. Options include ignoring the extra measurements or generating a run-time error.

**Note**  The type for measurement and limit values must be double-precision, 64-bit floating-point values, and cannot be signed or unsigned 64-bit integers.

### 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. When 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 and 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**—The string value. Usually, you set this value in the code module.
- **Step.Limits.String, StringExpr, and UseStringExpr**—The expected string for the string comparison.
- **Step.Comp**—The type of comparison, such as Ignore Case.
- **Step.CompExpr**—The comparison operation using an expression.
- **Step.UseCompExpr**—The step uses the expression to compare the string values.
Step.InBuf—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 maintains 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—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 when 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. Select Configure»Adapters from the menu bar and remove the checkmark from the checkbox in the Hidden column to display the Sequence Adapter in the Adapter ring control.
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. When the sequence fails, the Sequence Adapter sets the Sequence Call step status to Failed. A sequence fails when 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. When a run-time error occurs in the subsequence, the Sequence Adapter sets the step status to Error.

When you configure a Sequence Call step to use an expression to determine the names of the sequence and sequence file, you must use the Load Prototype button on the Module tab of the Step Settings pane of a Sequence Call step to load prototype information to configure the parameters. The order of the parameters in the prototype must match the order of the parameters in the sequence TestStand calls.

If the prototype you load does not specify all the parameters for a sequence TestStand calls, TestStand passes the default parameters values for the sequence for the unspecified parameters. If the prototype you load specifies additional parameters that the sequence TestStand calls does not specify, TestStand passes the additional parameters to the sequence.

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 when 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**—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 when a condition is met.

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

- **Step.ConditionExpr**—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 when the condition the preceding If or Else If step defines 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 when a condition is met and the conditions the preceding If step and any preceding Else If step define are not met.

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

- **Step.ConditionExpr**—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** — The expression the step evaluates before executing the steps within the block the first time. The expression typically initializes a count variable.
- **Step.ConditionExpr** — The expression that must evaluate to `True` for the steps within the For block to execute.
- **Step.IncrementExpr** — The expression the step evaluates after each execution of the steps within the block. The expression typically increments a count variable.
- **Step.CustomLoop** — 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** — The expression that determines the array over which the loop iterates.
- **Step.ArrayElementExpr** — The expression that determines the variable into which to store the current element of the array during each iteration of the loop.
- **Step.OffsetExpr** — The expression that determines the variable into which to store the current offset of the array during each iteration of the loop.
- **Step.SubscriptExpr** — 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**—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**—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**—The expression that determines which Case block within the Select block executes.
Chapter 4  Built-In Step Types

**Case**

Use a Case step, shown at left, to define a block of steps within a Select block that executes when 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**—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. When the step cannot evaluate the expression or when 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**—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**—The one-based index of the button you select.
- **Step.Result.Response**—The response text the user entered.
- **Step.TitleExpr**—The expression for the string that appears as the title of the message popup dialog box.
- **Step.MessageExpr**—The expression for the string that appears as the text message in the message popup dialog box.
- **Step.MessageFontData**—The font for the text message in the message popup dialog box.
- **Step.Button1Label, Button2Label, Button3Label, Button4Label, Button5Label, and Button6Label**—The expression for the label text for each button.
• **Step.ButtonFontData**—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**—The number of visible text lines in the response text box.

• **Step.MaxResponseLength**—The maximum number of characters the user can enter in the response text box control.

• **Step.RespFontData**—The font for the response text box control in the message popup dialog box.

• **Step.DefaultResponseExpr**—The initial text string the step displays in the response text box control.

• **Step.FileData**—The step displays 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**—The index of the button that activates automatically after a timeout elapses. A value of zero indicates no timeout occurs.

• **Step.TimeToWait**—The number of seconds before the button **Step.TimerButton** specifies activates.

• **Step.Position.Top** and **Step.Position.Left**—The location of the message popup dialog box when **CenterDialog** is **False**.

• **Step.CenterDialog**—The message popup dialog box appears in the center of the screen.

• **Step.Modal**—The message popup dialog box is modal to the TestStand application.

• **Step.Floating**—The message popup dialog box appears on top of the TestStand application.

• **Step.CtrlArrangement**—The order for the controls in the message popup dialog box.

• **Step.ButtonLocation**—Specifies whether to display the buttons on the bottom or side of the message popup dialog box.
• **Step.ButtonAlignment**—Specifies whether to align the buttons in the center, left, right, top, or bottom of the message popup dialog box.

• **Step.ResizeDialog**—Specifies whether the message popup dialog box is resizable.

### Call Executable

Use Call Executable steps, shown at left, to launch an application, open a file, run a system command, or call a script. When you specify the name of a non-executable file, the step launches the file using the default application associated with the extension of the file. For example, you can execute a Python or Perl script file or open a text file by specifying the non-executable file directly.

You can use expressions to specify the path to the executable and to configure the working directory. You can use an absolute path to the working directory, use the executable directory, use the sequence file directory, or use the current working directory of the TestStand application.

You can configure whether the step waits for the call to exit or waits for a specified amount of time. When a step waits for the call, you can configure the standard input passed to the call and where to store the standard output and error the call generates.

You can configure the step to stream a file as input or evaluate a string or an expression as input. You can store the output and error information in a file on the local computer or on the remote computer when you call the executable on a remote computer. TestStand overwrites the file if it already exists. Alternatively, you can use expressions to store the output and error information in TestStand variables or properties. When you store the information in a variable or property, you can use Additional Results to include the information in the report TestStand generates.

The final status of a Call Executable step can depend on whether the step waits for the executable to exit. When the step does not wait for the executable to exit, the step always sets the step status to **Done**. If a timeout occurs while the step is waiting for the executable to exit, the step sets the status to **Error**. When the step waits for the executable to exit and a timeout does not occur, the step 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. When you set the Exit Code Status Action ring control to **No Action**, the step always sets the step status to **Done**. Otherwise, you can choose to set the step status to **Failed** based on the value of the exit code—less than zero, greater than...
zero, equal to zero, or not equal to zero. You can also choose to ignore or set the step status to Error or Failed when the executable you call returns standard error information.

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

Refer to the Calling Scripting Languages from TestStand topic in the NI TestStand Help for more information about using a Call Executable step to call a script.

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

- **Step.Result.ExitCode**—The exit code the executable returns.
- **Step.Executable**—The pathname of the executable to launch.
- **Step.ExecutableExpr**—The expression for the pathname of the executable to launch.
- **Step.SpecifyExeByExpr**—Specifies whether the step uses the Step.Executable property or the Step.ExecutableExpr property to determine the executable to launch.
- **Step.ExecutableCalled**—The executable the step called.
- **Step.Arguments**—The expression for the argument string the step passes to the executable.
- **Step.WaitCondition**—Specifies how the step waits for the executable to exit before completing.
- **Step.TimeToWait**—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 whether to terminate the executable process when the execution terminates or aborts while waiting for the executable.
- **Step.StoreProcessHandle**—When this property is True, the step stores the Microsoft Windows process handle in a variable or property.
- **Step.ProcessHandleExpr**—The expression that determines the variable or property in which the step stores the Windows process handle for the executable.
- **Step.ProcessHandle**—Default property for the value of the Step.ProcessHandleExpr property to store the Windows process handle for the executable.
• **Step.ExitCodeStatusAction**—The condition the step evaluates for the exit code the executable returns to set the step status to Failed.

• **Step.ExitCodeErrorAction**—Specifies the condition the step evaluates for the exit code the executable returns to set the step status to Error.

• **Step.RemoteSettings**—Contains the following settings for calling the executable on a remote computer:
  - **Enabled**—TestStand calls the executable on a remote computer.
  - **Host**—The computer name or IP address of the remote computer. The Step.RemoteSettings.HostByExpr property specifies whether the step interprets this property value as an expression or as a string.
  - **HostByExpr**—When this setting is True, the step interprets the Step.RemoteSettings.Host property value as an expression. When this setting is False, the step interprets the value as a string.
  - **Port**—The port number the remote host server application uses.
  - **Password**—The password configured on the remote host server application. The Step.RemoteSettings.PasswordByExpr property specifies whether the step interprets this property value as an expression or as a string.
  - **PasswordByExpr**—When this setting is True, the step interprets the Step.RemoteSettings.Password property value as an expression. When this setting is False, the step interprets the value as a string.

• **Step.StdInput**—Contains the following standard input settings:
  - **Type**—A value that specifies the method for passing standard input to the call. The valid values are 0 = No Input, 1 = String, 2 = Expression, 3 = File on Local Machine, and 4 = File on Remote Machine.
  - **Source**—The standard input string, expression, or file pathname. The Step.StdInput.IsExpr property specifies whether the step interprets a file pathname value in this property as an expression or as a string.
  - **IsExpr**—When this setting is True, the step interprets a file pathname value in the SourceStep.StdInput.Source property as an expression. When this setting is False, the step interprets the value as a string.
• **Step.StdOutput**—Contains the following standard output settings:
  
  – **Dest**—A value that specifies the method for retrieving standard output from the call. The valid values are 0 = Ignore, 1 = Store in Variable/Property, 2 = Save to Local File, and 3 = Save to Remote File.
  
  – **Expr**—The standard output variable/property or file pathname. The Step.StdOutput.IsExpr property specifies whether the step interprets a file pathname value in this property as an expression or as a string.
  
  – **IsExpr**—When this setting is True, the step interprets a file pathname value in the SourceStep.StdOutput.Source property as an expression. When this setting is False, the step interprets the value as a string.
  
  – **Text**—Default property for the value of the Step.StdOutput.Expr property when the Step.StdOutput.Dest property specifies to store the standard output to a variable or property.

• **Step.WorkingDir**—Contains the following working directory settings:
  
  – **Source**—A value that specifies the method to determine the working directory. The valid values are 0 = Use Current Working Directory, 1 = Use Executable Directory, 2 = Use Sequence File Directory, and 3 = Specific Directory.
  
  – **Expr**—A specific working directory. The Step.WorkingDir.IsExpr property specifies whether the step interprets this property value as an expression or as a string.
  
  – **IsExpr**—When this setting is True, the step interprets the working directory path in the SourceStep.WorkingDir.Expr property as an expression. When this setting is False, the step interprets the path as a string.

• **Step.StdError**—Contains the following standard error settings:
  
  – **Dest**—A value that specifies the method for retrieving standard error from the call. The valid values are 0 = Ignore, 1 = Store in Variable/Property, 2 = Save to Local File, and 3 = Save to Remote File.
  
  – **Expr**—The standard error variable/property or file pathname. The Step.StdError.IsExpr property specifies whether the step interprets a file pathname value in this property as an expression or as a string.
– **IsExpr**—When this setting is True, the step interprets a file pathname value in the `SourceStep.StdError.Expr` property as an expression. When this setting is False, the step interprets the value as a string.

– **Text**—Default property for value of the `Step.StdError.Expr` property when the `Step.StdError.Dest` property specifies to store the standard error in a variable or property.

- **Step.Action**—Specifies how TestStand responds when the standard error is not empty. The valid values are 0 = No Action, 1 = Set Step Status to Failed, 2 = Set Step Status to Error.

**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. The Property Loader step type supports only ANSI file formats. 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**—The computer name or IP address of the remote computer. The `Step.RemoteHostByExpr` property specifies whether the step interprets this property value as an expression or as a string.

- **Step.RemoteHostByExpr**—When this property is True, the step interprets the `Step.RemoteHost` property value as an expression. When this property is False, the step interprets the value as a string.

- **Step.FTPUsername**—The login name to use when connecting to the server.

- **Step.FTPPassword**—The password to use when connecting to the server.

- **Step.FilesToFTP**—The local and remote file paths, the direction to transfer the file, and whether to overwrite a file when it exists.


Additional Results
Use an Additional Results step, shown at left, to specify values to add to the result list of a step. 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. You can configure TestStand to automatically include additional results in reports and when logging results to a database. 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. Refer to Chapter 7, Effectively Using LabVIEW with TestStand, of the Using LabVIEW and LabWindows/CVI with TestStand manual for more information about how to best use LabVIEW features in a TestStand system.
Module Adapters

The TestStand Engine uses module adapters to invoke code modules that 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.

Adapters specific to an application development environment (ADE) 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 a TestStand User Interface.

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 Microsoft Windows system directories, and you can add custom directories to the list. Use relative paths when 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, 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
Chapter 5  Module Adapters

- `<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`

When you list a directory and a subdirectory within that directory, TestStand performs a double search. You might want to use a double search only when 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`. You can use the Force File to Install option on the Distributed Files Tab of the TestStand Deployment Utility to overwrite existing files. Refer to Chapter 14, Deploying TestStand Systems, for more information about the deployment utility.

LabVIEW Adapter

Use the LabVIEW Adapter to call VIs with a variety of connector panes. The VIs can be on disk, in LLBs, in LabVIEW projects, or in LabVIEW packed project libraries. Use the LabVIEW Module tab on the Step Settings pane to configure calls to VIs.
**Note**  You must have LabVIEW 2010 or later to use LabVIEW packed project libraries in TestStand.

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 and LabWindows/CVI with TestStand* manual for more information about using the LabVIEW Adapter, supported data types, and tutorials that use the adapter. Refer to Chapter 7, *Effectively Using LabVIEW with TestStand*, of the *Using LabVIEW and LabWindows/CVI with TestStand* manual for more information about how to best use LabVIEW features in a TestStand system.

**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 LabVIEW and 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.
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.

When you launch Visual Studio from TestStand, Visual Studio runs with the same privileges you used to run TestStand. If you launch TestStand while logged in as a user with standard privileges and then you launch Visual Studio from TestStand, you cannot execute tasks in Visual Studio that require administrator privileges.

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.

**Accessing the TestStand API in Visual Studio**

You can use the `#import` compiler directive to automatically generate C++ classes for Visual Studio to access the objects defined in TestStand type libraries. Refer to the *Microsoft Visual C++/#import* topics in the *NI TestStand Help* for more information about using `#import` to access the TestStand API in Visual Studio.

**TestStand Include Directories**

TestStand adds TestStand API directories to the list of include directories that Visual Studio uses to locate files.

In Visual Studio 2008 or earlier, if TestStand directories do not appear in the Include Directories list of the VC++ Directories settings in the Options dialog box, you can manually add them. You can also add them by exiting all running copies of Visual Studio, selecting **Start»All Programs»National Instruments»TestStand»TestStand Version Selector** to run the TestStand Version Selector utility, selecting the current version of TestStand, and clicking the **Make Active** button.

In Visual Studio 2010 or later, the VC++ Directories settings in the Options dialog box are deprecated. Instead, use the **Enable TestStand Include Directories** option on the NI TestStand property page of the Project Properties dialog box for a C++ project to specify whether to use TestStand include directories.
Using DLLs

You can call LabVIEW, Microsoft Foundation Class (MFC), and subordinate DLLs from TestStand. If a DLL contains export information, TestStand displays parameter information on the Module Tab on the Step Settings pane.

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. When the TestStand step that calls the DLL preloads the DLL, TestStand ensures that the DLL loads in an STA thread. However, when 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 MFC Library places several requirements on DLLs that use the DLL version of the MFC run-time library. When you call a DLL that includes MFC functions, verify that the DLL meets these requirements. Also, when 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 XP SP1 or earlier) The current working directory of the application
3. The `Windows\System32` and `Windows\System` directories
4. The `Windows` directory
5. **(Windows XP SP2 or later)** The current working directory of the application
6. The directories listed in the PATH environment variable

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

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

**Reading Parameter Information**

When a DLL contains export information or when 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. When the adapter cannot determine parameter information, you must enter the parameter information manually.

Refer to Chapter 17, *Adding Type Libraries to LabWindows/CVI DLLs*, of the *Using LabVIEW and 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, when 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.

When 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.

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 directly from TestStand into a supported version of Visual Studio, you must configure the step to use the C/C++ DLL Adapter. When 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.

Note  You must attach Visual Studio to the TestStand process correctly depending on the type of code module you want to debug. For C++ code modules, National Instruments recommends using only native debugging when attaching Visual Studio.
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

<table>
<thead>
<tr>
<th>ADE Command for Stepping Out</th>
<th>Result in TestStand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finish Function or Step Out</td>
<td>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.</td>
</tr>
<tr>
<td>Step Into or Step Over</td>
<td>When you use this command on the last executable statement of the function, TestStand suspends execution on the next step in the sequence.</td>
</tr>
<tr>
<td>Continue</td>
<td>TestStand does not suspend execution when the function call returns.</td>
</tr>
</tbody>
</table>

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

When you launch Visual Studio from TestStand, Visual Studio runs with the same privileges you used to run TestStand. If you launch TestStand while logged in as a user with standard privileges and then you launch Visual Studio from TestStand, you cannot execute tasks in Visual Studio that require administrator privileges.

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

**Debugging LabVIEW Shared Libraries (DLLs)**

To build a debuggable shared library in the LabVIEW development environment, you must configure the shared library to enable debugging in the build specification for the shared library before building the shared library. Using the LabVIEW development environment, you can then connect to the TestStand application process that loads and calls the VIs in the debuggable shared library. Refer to the LabVIEW Help for more information about building and debugging applications and shared libraries.
.NET Adapter

Use the .NET Adapter to call .NET assemblies written in any .NET-compliant language, such as C#, C++/CLI, or Microsoft Visual Basic .NET.

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. The .NET Adapter also provides the following functionality for integrating .NET code with TestStand:

- In versions of TestStand earlier than TestStand 2010, you could call code modules with the .NET Adapter even when the prototype in the assembly did not match the prototype you specified for the step. In TestStand 2010, the .NET Adapter and the TestStand Sequence Analyzer return an error when the prototype does not match exactly. However, for backward compatibility, the .NET Adapter uses a similar algorithm as in previous versions of TestStand to try to run the code modules despite the error. National Instruments does not recommend relying on this behavior. National Instruments recommends that you use the sequence analyzer and update the prototypes of .NET steps as needed when the prototypes in code modules change.

- For a struct, you can optionally define a corresponding TestStand data type with the same fields as the struct and use instances of the data type instead of a .NET version of the struct wherever the struct is used with the adapter.

- You can directly call .NET string class members on TestStand strings using the `System.String` class, which `mscorlib.dll` of the .NET Framework defines.

- You can store any .NET variable in a TestStand object reference variable, which is useful for storing large variables, such as arrays, without having to create a TestStand equivalent.

- You can use the `PropertyObject.SetValInterface` and `PropertyObject.GetValInterface` methods of the TestStand API to programmatically store and retrieve .NET objects in TestStand object reference variables.

- You can create and edit .NET code modules in a supported version of Visual Studio directly from TestStand.
You can step into .NET code modules in a supported version of the Visual Studio debugger directly from a TestStand sequence you are debugging.

**Note** When you launch Visual Studio from TestStand, Visual Studio runs with the same privileges you used to run TestStand. If you launch TestStand while logged in as a user with standard privileges and then you launch Visual Studio from TestStand, you cannot execute tasks in Visual Studio that require administrator privileges.

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

**Generics**

The .NET Adapter supports all fully-specified generic types that the public members of an assembly use. Thus, the adapter supports any generic type used as a parameter, return value, property, or field type in a public member. For example, you can call an assembly with a method that takes the `List<Int32>` type as a parameter and you can create instances of the type and call members on the type.

**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.

When you use a supported version of Visual Studio 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.

When 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 and native 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**.
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

<table>
<thead>
<tr>
<th>Visual Studio Command for Stepping Out</th>
<th>Result in TestStand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step Out</td>
<td>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.</td>
</tr>
<tr>
<td>Step Into or Step Over</td>
<td>When you use this command on the last executable statement of the function, TestStand suspends execution on the next step in the sequence.</td>
</tr>
<tr>
<td>Continue</td>
<td>TestStand does not suspend execution when the function call returns.</td>
</tr>
</tbody>
</table>

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

Note When you use LabWindows/CVI to debug a DLL in the TestStand process, you cannot debug a .NET assembly at the same time. When 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

With versions of the .NET Framework earlier than version 4.0, an application can load only one version of the .NET Common Language Runtime (CLR) into memory. By default, TestStand cannot call assemblies that require a version of the .NET CLR later than the version the TestStand process loads, although assemblies written for earlier versions of the .NET CLR generally work in a later version of the .NET CLR. The .NET Framework 2.0, 3.0, and 3.5 all use the .NET CLR version 2.0, which is the version TestStand loads by default.
To call assemblies from TestStand that require a later version of the .NET CLR, such as .NET CLR 4.0 for the .NET Framework 4.0, and force a TestStand application to use the later version, create a configuration file in the same directory as the application executable. For example, to force the sequence editor to load .NET CLR 4.0, create a file called *seqedit.exe.config* with the following content in the `<TestStand>\Bin` directory:

```xml
<?xml version="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    </assemblyBinding>
  </runtime>
  <startup useLegacyV2RuntimeActivationPolicy="true">
    <supportedRuntime version="v4.0" />
  </startup>
</configuration>
```

**Note** The `useLegacyV2RuntimeActivationPolicy="true"` attribute is required for the TestStand Engine to call .NET Framework 4.0 assemblies. Therefore, even if an application is directly targeting the .NET Framework 4.0, you must still create a configuration file that contains this attribute.

**Accessing the TestStand API in Visual Studio**

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

To add a reference to the TestStand 4.0 or later API assembly in Visual Studio, 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.

If the TestStand Interop assemblies do not appear in the Add Reference dialog box, exit all running copies of Visual Studio, select **Start» All Programs»National Instruments»TestStand»TestStand Version Selector** to run the TestStand Version Selector utility, select the current version of TestStand, and click the **Make Active** button.
Refer to the C# and Microsoft Visual Basic .NET topics in the NI TestStand Help for more information about using the TestStand API in Visual Studio from .NET applications.

**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. When 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, when 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 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 TestStand 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.

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 7/Vista. However, if you installed the HTBasic 9.0 development environment under Windows 7/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.

When 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 when 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, the TestStand data type of the parameter, the default value of the parameter, 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.
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. When 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. When you want to access properties from the client sequence context, you must pass the PropertyObject objects or their values as parameters to

<table>
<thead>
<tr>
<th>Type of Path</th>
<th>Where Found When You Edit</th>
<th>Where Found When You Execute</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative</td>
<td>In the TestStand search paths you configure on the client (local) computer</td>
<td>In the TestStand search paths you configure on the server (remote) computer</td>
<td>Transmit.seq</td>
</tr>
<tr>
<td>Absolute</td>
<td>On the client (local) computer</td>
<td>On the server (remote) computer</td>
<td>C:\Projects\Transmit.seq</td>
</tr>
<tr>
<td>Network</td>
<td>On the computer the network path specifies</td>
<td>On the computer the network path specifies</td>
<td>\Remote\Projects\Transmit.seq</td>
</tr>
</tbody>
</table>
the remote sequence. You can use the TestStand API to access properties within a property object.

You must properly configure a remote computer and the TestStand server application on the remote computer when 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. When you use the TestStand API on the server computer to access objects on the client computer, you must also configure Windows system security on the local client.

**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.

**Note** Enabling the Show the System Tray Icon While the TestStand Remote System is Active on this Machine option requires that a user be logged in at the physical computer console when the remote engine application launches. You can automatically log on a user each time a computer reboots by setting registry keys. Refer to Microsoft Knowledge Base article 315231 at support.microsoft.com/kb/315231 for more information about activating automatic logon.
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 on a Remote Server

You must configure the Distributed Component Object Model (DCOM) settings and Windows firewall settings to allow users to access and launch the TestStand server remotely.

**DCOM Settings for Remote Server**

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

*(Windows 7/Vista)* Windows 7/Vista launches a User Account Control elevation prompt for you to manually resolve when you enable this option.

**Note** When you enable the Allow All Users Access From Remote Machines option, a user must be logged in at the physical computer console during remote sequence execution. In addition, the user account must be an administrator account. You can automatically log on a user each time a computer reboots by setting registry keys. Refer to Microsoft Knowledge Base article 315231 at support.microsoft.com/kb/315231 for more information about activating automatic logon.

When you enable the Allow All Users Access From Remote Machines option, TestStand configures the DCOM security permissions for you in the following ways:

- TestStand grants the Remote Launch and Remote Activation privileges to the Everyone and ANONYMOUS LOGON user names.
- TestStand changes the computer Component Object Model (COM) security limits to grant the Remote Launch, Remote Activation, and Remote Access privileges to the Everyone and ANONYMOUS LOGON user names.

In addition, when you enable the Allow All Users Access From Remote Machines option, the remote engine application overrides the application DCOM authentication level setting and does not authenticate connections. When you disable this option, TestStand revokes all the privileges listed previously, even if they were granted before the option was enabled.
When you do not enable the Allow All Users Access From Remote Machines option, complete the following steps to configure the DCOM security permissions for the server on a remote computer.

1. Log in as a user with administrator privileges.
2. 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.
3. On 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 and complete the following steps to set permissions.
   a. Click the Edit Limits button in the Access Permissions section to launch the Access Permission dialog box.
   b. Click Add to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the ANONYMOUS LOGON user.
   c. *(Windows 7/Vista)* Click OK to close the Select Users or Groups dialog box.
      *(Windows XP)* Click OK to close the Select Users, Computer, Groups dialog box.
   d. Select the user you added and enable Remote Access in the Permissions section.
   e. Click OK to close the Access Permission dialog box.
   f. *(Windows 7/Vista)* Click the Edit Limits button in the Launch and Activation Permissions section to launch the Launch and Activation Permission dialog box.
      *(Windows XP)* Click the Edit Limits button in the Launch and Activation Permissions section to launch the Launch Permission dialog box.
g. Click Add to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the ANONYMOUS LOGON user.

h. (Windows 7/Vista) Click OK to close the Select Users or Groups dialog box.
   (Windows XP) Click OK to close the Select Users, Computer, Groups dialog box.

i. Select the user you added and enable Remote Launch and Remote Activation in the Permissions section.

j. (Windows 7/Vista) Click OK to close the Launch and Activation Permission dialog box.
   (Windows XP) 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. On 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 General tab of the NI TestStand Remote Engine Properties dialog box, set the Authentication Level to None if the user is not a domain account.

11. On the Identity tab of the NI TestStand Remote Engine Properties dialog box, select the This user option and enter a user name and password or select the The interactive user option.
   - If you select the This user option, you must disable the Show the System Tray Icon While the TestStand Remote System is Active on this Machine option in the Station Options dialog box.
   - If you select the The interactive user option, a user must be logged in at the physical computer console during remote sequence execution. You can automatically log on a user each time a computer reboots by setting registry keys. Refer to Microsoft Knowledge Base article 315231 at support.microsoft.com/
kb/315231 for more information about activating automatic logon.

**Note** Regardless of whether you select the This user option or The interactive user option, the user account used on the remote computer must be an administrator account.

12. Click the **Security** tab of the NI TestStand Remote Engine Properties dialog box and complete the following steps to set permissions.

a. *(Windows 7/Vista)* Select the **Customize** option and click the **Edit** button in the Launch and Activation Permissions section to launch the Launch and Activation Permission dialog box.

   *(Windows XP)* Select the **Customize** option and click the **Edit** button in the Launch and Activation Permissions section to launch the Launch Permission dialog box.

b. Click **Add** to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the **ANONYMOUS LOGON** user.

c. *(Windows 7/Vista)* Click **OK** to close the Select Users or Groups dialog box.

   *(Windows XP)* Click **OK** to close the Select Users, Computer, Groups dialog box.

d. Select the user you added and enable **Remote Launch** and **Remote Activation** in the Permissions section.

e. *(Windows 7/Vista)* Click **OK** to close the Launch and Activation Permission dialog box.

   *(Windows XP)* Click **OK** to close the Launch Permission dialog box.

f. Select the **Customize** option and click the **Edit** button in the Access Permissions section to launch the Access Permission dialog box.

g. Click **Add** to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the **ANONYMOUS LOGON** user.

h. *(Windows 7/Vista)* Click **OK** to close the Select Users or Groups dialog box.

   *(Windows XP)* Click **OK** to close the Select Users, Computer, Groups dialog box.

i. Select the user you added and enable **Remote Access** in the Permissions section.
13. Click **OK** to close the NI TestStand Remote Engine Properties dialog box.

**Windows Firewall Settings for Remote Server**

Use the following information to configure the Windows firewall on the server computer to allow the TestStand client to access the `REngine.exe` application.

**Windows 7 Firewall Settings**

Complete the following steps to configure the Windows firewall on the server computer.

1. Log in as a user with administrator privileges.

2. Navigate to **Windows Firewall** on the Windows Control Panel and click **Allow a program or feature through Windows Firewall** on the left panel of the Windows Firewall window to launch the **Allowed Programs** window.

3. Complete the following steps to add exceptions for the `REngine.exe` application with the firewall enabled.
   a. Click the **Allow another program** button to launch the Add a Program dialog box.
   b. Click **Browse** and select `<TestStand>\Bin\REngine.exe`.
   c. Click **Add** to close the Add a Program dialog box.
   d. Click **OK** to close the Allowed Programs window.

4. Click **Advanced settings** on the left panel of the Windows Firewall window to launch the Windows Firewall with Advanced Security window.

5. Complete the following steps to specify a security exception for DCOM.
   a. Click **Inbound Rules** on the left panel of the Windows Firewall with Advanced Security window to display the list of inbound rules.
   b. Click **New Rule** on the Actions pane to launch the New Inbound Rule Wizard.
   c. Select **Port** and click the **Next** button.
   d. Select **TCP**, select **Specific local ports**, enter `135` in the Specific local ports control, and click **Next**.
   e. Select **Allow the connection** and click **Next**.
f. Ensure that the Domain, Private, and Public options have checkmarks and click **Next**.
g. In the Name control, enter **DCOM** and click **Finish** to close the New Inbound Rule Wizard.

**Windows Vista Firewall Settings**

Complete the following steps to configure the Windows firewall on the server computer.

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.
   c. Click **Browse** and select `<TestStand>\Bin\REngine.exe`.
   d. Click **OK** to close the Add a Program dialog box.
   e. Click the **Add port** button to launch the Add a Port dialog box.
   f. In the **Name** control, type **DCOM**.
   g. In the **Port Number** control, type **135**.
   h. Select the **TCP** option and click **OK** to close the Add a Port dialog box.
4. Click **OK** to close the Windows Firewall Settings dialog box.

**Windows XP Service Pack 2 Firewall Settings**

Complete the following steps to configure the Windows firewall on the server computer.

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.

c. Click **Browse** and select `<TestStand>\Bin\REngine.exe`.

d. Click **OK** to close the Add a Program dialog box.

e. Click the **Add Port** button to launch the Add a Port dialog box.

f. In the **Name** control, type **DCOM**.

g. In the **Port Number** control, type **135**.

h. Select the **TCP** option and click **OK** to close the Add a Port dialog box.

4. Click **OK** to close the Windows Firewall dialog box.

### Setting Windows System Security on a Local Client

You must configure DCOM settings and Windows firewall settings on the local client computer when you use the TestStand API on the remote server to access objects on the local client. For example, when you pass a property object as an argument to a sequence running on a TestStand remote server and that sequence attempts to access subproperties of the parameter, the remote server must be able to access the property object on the client computer.

#### DCOM Settings for Local Client

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. 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.

3. On 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 and complete the following steps to set permissions.

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. If **ANONYMOUS**
LOGON does not appear in the user name list, click the **Add** button to add it.

c. Click **OK** to close the Access Permissions dialog box.

You must also disable DCOM authentication for the client application if it is not the TestStand Sequence Editor (*SeqEdit.exe*) or one of the TestStand User Interfaces named *testexec.exe*. For client applications built with LabVIEW, disable DCOM authentication for the application by adding the following line to the INI file associated with the application, as in *yourclient.ini* where *yourclient.exe* is the name of your application:

```
ole.AuthnLevel=1
```

For all other client applications, disable DCOM authentication for the application by adding the following registry entry, where *yourclient.exe* is the name of your application:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AppID\yourclient.exe]
"AppID" ="{C31FD07F-DEAC-4962-9BBF-092F0F3BFF3C}"
```

### Windows Firewall Settings for Local Client

Use the following information to configure the Windows firewall on the client computer to allow the *REngine.exe* application to access the TestStand client.

#### Windows 7 Firewall Settings

Complete the following steps to configure the Windows firewall on the client computer.

1. Log in as a user with administrator privileges.
2. Navigate to **Windows Firewall** on the Windows Control Panel and click **Allow a program or feature through Windows Firewall** on the left panel of the Windows Firewall window to launch the Allowed Programs window.
3. Complete the following steps to add exceptions for the client application with the firewall enabled.
   a. Click the **Allow another program** button to launch the Add a Program dialog box.
   b. Click **Browse** and select the client application, such as `<TestStand>\Bin\SeqEdit.exe`.
   c. Click **Add** to close the Add a Program dialog box.
   d. Click **OK** to close the Allowed Programs window.
4. Click **Advanced settings** on the left panel of the Windows Firewall window to launch the Windows Firewall with Advanced Security window.

5. Complete the following steps to specify a security exception for DCOM.
   a. Click **Inbound Rules** on the left panel of the Windows Firewall with Advanced Security window to display the list of inbound rules.
   b. Click **New Rule** on the Actions pane to launch the New Inbound Rule Wizard.
   c. Select **Port** and click the **Next** button.
   d. Select **TCP**, select **Specific local ports**, enter 135 in the Specific local ports control, and click **Next**.
   e. Select **Allow the connection** and click **Next**.
   f. Ensure that the Domain, Private, and Public options have checkmarks and click **Next**.
   g. In the Name control, enter **DCOM** and click **Finish** to close the New Inbound Rule Wizard.

### Windows Vista Firewall Settings

Complete the following steps to configure the Windows firewall on the client computer.

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 client application with the firewall enabled.
   a. Click the **Exceptions** tab.
   b. Click the **Add program** button to launch the Add a Program dialog box.
   c. Click **Browse** and select the client application, such as `<TestStand>\Bin\SeqEdit.exe`.
   d. Click **OK** to close the Add a Program dialog box.
   e. Click the **Add port** button to launch the Add a Port dialog box.
   f. In the Name control, type **DCOM**.
g. In the Port Number control, type 135.

h. Select the TCP option and click OK to close the Add a Port dialog box.

4. Click OK to close the Windows Firewall Settings dialog box.

Windows XP Service Pack 2 Firewall Settings

Complete the following steps to configure the Windows firewall on the client computer.

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 client application with the firewall enabled.
   a. Click the Exceptions tab.
   b. Click the Add Program button to launch the Add a Program dialog box.
   c. Click Browse and select the client application, such as <TestStand>\Bin\SeqEdit.exe.
   d. Click OK to close the Add a Program dialog box.
   e. Click the Add Port button to launch the Add a Port dialog box.
   f. In the Name control, type DCOM.
   g. In the Port Number control, type 135.
   h. Select the TCP option and click OK to close the Add a Port dialog box.

4. Click OK to close the Windows Firewall dialog box.
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 Microsoft 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 unit under test (UUT) number, a step name, a step result, and a measurement.

**Table 6-1. Example Database Table**

<table>
<thead>
<tr>
<th>UUT_NUM</th>
<th>STEP_NAME</th>
<th>RESULT</th>
<th>MEAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>20860B456</td>
<td>TEST1</td>
<td>PASS</td>
<td>0.5</td>
</tr>
<tr>
<td>20860B456</td>
<td>TEST2</td>
<td>PASS</td>
<td>(NULL)</td>
</tr>
<tr>
<td>20860B123</td>
<td>TEST1</td>
<td>FAIL</td>
<td>0.1</td>
</tr>
</tbody>
</table>
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) 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.

---

### Table 6-1. Example Database Table (Continued)

<table>
<thead>
<tr>
<th>UUT_NUM</th>
<th>STEP_NAME</th>
<th>RESULT</th>
<th>MEAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>20860B789</td>
<td>TEST1</td>
<td>PASS</td>
<td>0.3</td>
</tr>
<tr>
<td>20860B789</td>
<td>TEST2</td>
<td>PASS</td>
<td>(NULL)</td>
</tr>
</tbody>
</table>
Refer to [www.microsoft.com](http://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).

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 for more information about using 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 Database Known Issues topic in the
NI TestStand Help for information about known issues when using DBMS
client software with TestStand. 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**—The UUT result TestStand is processing. When TestStand is processing a step or a subproperty, this property holds the UUT result that contains the step result or subproperty.

- **StepResult**—The step result TestStand is processing. When TestStand is processing a subproperty, this property holds the step result that contains the subproperty. When 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**—The subproperty of the step result TestStand is processing. When TestStand is not processing a subproperty, this property does not exist.

- **PropertyResultDetails**—Information about the subproperty of the step result TestStand is processing. When TestStand is not processing a subproperty, this container does not exist.

- **ExecutionOrder**—A numeric value TestStand increments after it processes each step result.
• **StartDate**—The date on which the UUT test began. This property is an instance of the DateDetails custom data type.

• **StartTime**—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**—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**—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 database table.

### Creating Default Result Tables with the Database Viewer

Select **Start»All Programs»National Instruments»TestStand»Tools»Database Viewer** to launch the TestStand Database Viewer 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. The Database Viewer application is located at `<TestStand>\Components\Tools\DatabaseView\DatabaseView.exe`.

**Note** You must set up the DBMS server and any required DBMS client software before using the Database Viewer application.

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.

Click the **Duplicate** button on the Schemas tab of the Database Options dialog box to copy an existing schema and then customize the 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:

   ```sql
   CREATE TABLE UUT_RESULT
   (
     IDNUMBER 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.

When 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 7/Vista) If you use the ODBC Administrator on Windows 7/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.
5. Select Access from the Database Management System ring control.
6. Click the Build button to launch the Data Link Properties dialog box.

7. Select Microsoft Jet 4.0 OLE DB Provider on the Provider tab of the Data Link Properties dialog box.

8. Click Next.

9. On the Connection tab, click Browse to launch the Select Access Database dialog box.

10. Using the Select Access Database dialog box, locate an Access database file (.mdb) and click Open to select the file.

11. In the Data Link Properties dialog box, click the Test Connection button to verify that you properly entered the required information.

12. 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.
5. 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.

6. Click the **Execute SQL Commands** button to create the default result tables.

7. Review the results of the SQL commands in the SQL History control to ensure that you created the tables successfully.

8. 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 results to the database.

### Report Generation Implementation

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

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

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 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 name, format, and content of reports. Refer to the *NI TestStand Help* for more information about the Report Options dialog box.
Using Reports

The Test UUTs and Single Pass entry points in the TestStand process models generate reports. The Test UUTs entry point generates a 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 content and format of the report and the names and locations of report files.

In the TestStand Sequence Editor, the Report pane of 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, Automatic Test Markup Language (ATML), HTML, or ASCII-text formats.

You can also use an external application to view reports. 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.

Using Expressions to Customize Reports

In addition to using options to determine the names and locations of report files, you can specify a custom expression with predefined macros on the Report File Pathname tab of the Report Options dialog box. TestStand evaluates the expression and macros at runtime for each UUT, which allows you to fully customize the filename of reports and the directory in which TestStand stores reports. For example, you can use expressions to save reports in a directory that uses the sequence filename, save reports in different directories based on execution status, and generate report filenames that include the serial number, user name, or execution status. You can use expressions to replicate all existing report filename options.
Table 6-2 lists common report options and the equivalent expression for the Sequential model using the client sequence file directory as the base directory.

<table>
<thead>
<tr>
<th>Current Report Option</th>
<th>Equivalent Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>New UUT Report File for Each UUT</td>
<td><code>&lt;ClientFileDir&gt;\Report_&lt;UUT&gt;.&lt;FileExtension&gt;</code></td>
</tr>
<tr>
<td>Prefix Sequence File Name to Report File Name</td>
<td><code>&lt;ClientFileDir&gt;\&lt;ClientFileName&gt;_Report.&lt;FileExtension&gt;</code></td>
</tr>
<tr>
<td>Add Date and Time to File Name</td>
<td><code>&lt;ClientFileDir&gt;\Report_&lt;FileDate&gt;&lt;FileTime&gt;.&lt;FileExtension&gt;</code></td>
</tr>
<tr>
<td>Force File Name to Be Unique</td>
<td><code>&lt;ClientFileDir&gt;\Report_&lt;Unique&gt;.&lt;FileExtension&gt;</code></td>
</tr>
<tr>
<td>Prefix Sequence File Name to Report File Name + New UUT Report File for Each UUT + Add Date and Time to File Name</td>
<td><code>&lt;ClientFileDir&gt;\&lt;ClientFileName&gt;_Report_&lt;UUT&gt;_&lt;FileDate&gt;&lt;FileTime&gt;.&lt;FileExtension&gt;</code></td>
</tr>
</tbody>
</table>

Table 6-3 lists example expressions for common tasks using the Sequential model.

<table>
<thead>
<tr>
<th>Task</th>
<th>Equivalent Expression</th>
</tr>
</thead>
</table>
| Save reports in a directory that uses the sequence filename | C:\<ClientFileName>\Report.<FileExtension>  
**Note**: TestStand overwrites existing files unless you use the `<Unique>` macro or enable the Append if File Already Exists option. |
| Save reports in different directories based on execution status | C:\<ClientFileName>\<UUTstatus>\Report.<FileExtension> 
**Note**: TestStand overwrites existing files unless you use the `<Unique>` macro or enable the Append if File Already Exists option. |
| Generate report filenames that include the UUT serial number | C:\<ClientFileName>\Report_<UUT>.<FileExtension> |
| Generate report filenames that include the user name | C:\<ClientFileName>\Report_<UserName>.<FileExtension> |
| Generate report filenames that include the execution status | C:\<ClientFileName>\Report_<UUTStatus>.<FileExtension> |

Refer to the *NI TestStand Help* for a list of supported macros you can use with expressions to customize reports.

**Note** The list of macros is for use only in the Specify Report File Path by Expression control on the Report File Pathname tab of the Report Options dialog box and not in any other TestStand expression controls.
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.

The ATML standard does not support batch reports. Therefore, the Batch process model generates only UUT reports when you select ATML as a report format.

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. When 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 report concurrently with the execution instead of waiting until the execution or UUT test completes. The final report TestStand generates is identical to the report the process model generates at the end of execution. You can use on-the-fly reporting for XML, HTML, and ASCII reports.

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

In addition to generating the report concurrently with execution, on-the-fly reporting periodically persists the current report to a temporary file based on the persistence interval the process model sequences specify. TestStand deletes the temporary file and saves the final 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.

When you enable the Conserve Memory and Only Display Latest Results option for HTML and ASCII reports, on-the-fly reporting periodically purges internal data structures. As a result, the 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 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 records each result.

**XML Reports**

TestStand generates two types of XML reports—XML reports that validate against the TestStand Report schema, which conforms to the XML W3C Schema, and ATML reports that validate against the Test Results and Session Information schema the ATML standard defines. By default, TestStand generates XML reports not ATML reports. You cannot modify the XML TestStand generates, the TestStand Report schema, or the ATML Test Results and Session Information report schema, but you can customize how TestStand XML and ATML reports appear when displayed in a browser. Additionally, you can write custom XML report generators to generate your own XML report.

The TestStand Report schema, `Report.xsd`, is located in the `<TestStand>\Components\Models\TestStandModels\TestStandModels directory. Refer to the *NI TestStand Help* for more information about the TestStand Report schema. The ATML Test Results and Session Information schema, `TestResults.xsd`, is located in the `<TestStand>\Components\Models\TestStandModels\ATML\Schemas directory.

Refer to the *NI TestStand Help* for more information about ATML.
Transforming XML Reports

Extensible Stylesheet Language Transformations (XSLT) is an XML-based language that transforms XML documents into other formats, such as a different XML structure, HTML, and text. Use XSLT to separate the formatting of XML data from the actual XML data itself. XSLT does not usually modify the existing XML file. Instead, XSLT generates a new output file based on the existing XML file. Therefore, you can apply multiple XSLT transforms to create multiple views for the same XML data. Refer to the World Wide Web Consortium (W3C) Web site, located at www.w3.org/TR/xslt, for more information about XSLT.

XSLT also usually uses JavaScript, VBScript, XPath, and XQuery technologies along with XSL to transform XML documents. Refer to the W3Schools Web site, located at www.w3schools.com, for more information about these technologies.

Associating Style Sheets with XML Reports

The <xml-stylesheet> node of an XML report contains the URL path of the style sheet to apply to the XML file when you view the file in a browser or a viewer. The following example uses the TestStand horizontal style sheet:

```xml
<?xml-stylesheet type="text/xsl" href="C:\Program Files\National Instruments\TestStand 2010\Components\Models\TestStandModels\StyleSheets\horizontal.xsl"?>
```

Refer to the NI TestStand Help for more information about the TestStand report style sheets and for examples of common report style sheet customizations.

You can use the Contents tab of the Report Options dialog box to select the style sheet to apply to an XML report. Refer to the NI TestStand Help for more information about the Report Options dialog box.
Example HTML Transformation

The following example XML file associates the samplexStylesheet.xsl style sheet:

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<?xml-stylesheet type="text/xsl" href="C:\samplexStylesheet.xsl" ?>
<source>
  <title>XSL tutorial</title>
  <author>John Smith</author>
</source>
```

The following example XSLT code transforms XML to HTML. The style sheet uses XPATH expressions to add the title and author from the XML file to the generated HTML output. The style sheet also contains required HTML formatting, such as table and border formats, to add to the generated HTML output.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version='1.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:output method="html"/>
  <xsl:template match="/">
    <head>
      <title>Generated HTML Output</title>
    </head>
    <body>
      <table border = "1">
        <tr>
          <td>Title</td>
          <td>Author</td>
        </tr>
        <tr>
          <td><xsl:value-of select="//title"/></td>
          <td><xsl:value-of select="//author"/></td>
        </tr>
      </table>
    </body>
  </xsl:template>
</xsl:stylesheet>
```
The previous example XSLT code generates the following HTML:

```html
<head>
  <title>Generated HTML Output</title>
</head>
<body>
  <table border="1">
    <tr>
      <td><b>Title</b></td>
      <td><b>Author</b></td>
    </tr>
    <tr>
      <td>XSL tutorial</td>
      <td>John Smith</td>
    </tr>
  </table>
</body>
```

The previous HTML code appears as follows when viewed in a browser:

![Table](image)

**Figure 6-2. Sample HTML Transform**

### Distributing XML Reports

You can distribute XML reports by selecting **Tools»Package XML/HTML Files for Distribution** to launch the XML Packaging Utility. You can also select **Start»All Programs»National Instruments»TestStand»Tools»Package XML or HTML Files for Distribution** to launch this tool. This utility copies the local XML report files you select, including related style sheets, image files, and referenced files, to a destination directory you specify so you can view the XML report on a computer that does not have TestStand installed. Refer to the *NI TestStand Help* for more information about the XML Packaging Utility.

### Displaying Measurement Data

XML reports might unexpectedly display a table of measurement data instead of a graph under either of the following conditions:

- The TSGraphControl.ocx ActiveX control, located in the `<TestStand Public>\Components\Tools\GraphControl` directory, is not installed or registered on the computer.
The My Computer security zone setting on the Security tab of the Internet Options dialog box on the Windows Control Panel is High. Internet Explorer security settings can prevent Web pages from using scripting to create and use an ActiveX control. When you view Web pages located on the computer, Internet Explorer uses the My Computer security zone setting. On Windows XP SP2, the default security level for the My Computer security zone is High, which prevents scripting from running locally. By default, the settings for the My Computer security zone are not visible in the Internet Options dialog box.

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

Refer to the National Instruments Web site at ni.com/info and enter the Info Code rdgxml to access the TestStand support document, *Graphs in XML TestStand Report Do Not Show in External Viewer*, for more information about altering the Windows registry to make the My Computer security zone setting visible in the Internet Options dialog box.
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 configure TestStand to enforce user privileges and to specify the location of the user manager configuration file.

**Note** The 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. Refer to the *Front-End Callbacks* section of Chapter 10, *Customizing Process Models and Callbacks*, for information about customizing the default TestStand login/logout implementation.

**Privileges**

The user manager stores user and group privileges as Boolean properties and organizes the privileges in the following categories:

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

• **Configure**—Privileges for editing process model files and configuring station options, users, adapters, application settings, and report, database logging, and model options.

• **Custom**—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 when the property value for the privilege is *True* or when 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 when 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 whether 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 whether 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` when 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.
You can customize TestStand User Interfaces, process models, callbacks, data types, step types, the Tools menu, the directory structure, and TestStand Sequence Analyzer rules. You can configure options for the TestStand 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. The *NI TestStand Help* includes information about the sequence analyzer. 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**

<table>
<thead>
<tr>
<th>Component</th>
<th>How to Customize</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Interfaces</td>
<td>TestStand includes full source code in several different programming languages so you can modify the user interfaces to meet specific needs.</td>
<td>Refer to Chapter 9, <em>Creating Custom User Interfaces</em>.</td>
</tr>
<tr>
<td>Process Models</td>
<td>TestStand includes fully customizable Sequential, Parallel, and Batch process models to meet specific testing needs.</td>
<td>Refer to Chapter 10, <em>Customizing Process Models and Callbacks</em>.</td>
</tr>
<tr>
<td>Callbacks</td>
<td>You can modify callback sequences to customize the operation of test stations.</td>
<td>Refer to Chapter 10, <em>Customizing Process Models and Callbacks</em>.</td>
</tr>
<tr>
<td>Data Types</td>
<td>You can customize copies of the standard data types or create and modify your own data types.</td>
<td>Refer to Chapter 11, <em>Type Concepts</em>, and Chapter 12, <em>Standard and Custom Data Types</em>.</td>
</tr>
</tbody>
</table>
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 Microsoft Windows 7/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 or later installs some files in different locations from previous versions of TestStand. Refer to the Directory Relocation section of the NI TestStand Version 4.2 Release Notes for more information about specific directories and files relocated in TestStand 4.1 or later. Refer to the National Instruments Web site at ni.com/info and enter the Info Code ex3ukj to locate the NI TestStand Version 4.2 Release Notes.

TestStand installs files in the following directories:

- `<TestStand>`—Located by default at C:\Program Files\National Instruments\TestStand on Windows 32-bit systems and at C:\Program Files (x86)\National Instruments\TestStand on Windows 64-bit systems.
- `<TestStand Public>`—Located by default at C:\Users\Public\Documents\National Instruments\TestStand on Windows 7/Vista and at C:\Documents and Settings\All Users\Documents\National Instruments\TestStand on Windows XP.

### Table 8-1. TestStand Customizable Components (Continued)

<table>
<thead>
<tr>
<th>Component</th>
<th>How to Customize</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step Types</td>
<td>You can customize copies of the standard step types or create and modify your own step types.</td>
<td>Refer to Chapter 11, <em>Type Concepts</em>, and Chapter 13, <em>Custom Step Types</em>.</td>
</tr>
<tr>
<td>Sequence Analyzer Rules</td>
<td>You can modify the built-in rules or create and modify your own rules.</td>
<td>Refer to the <em>NI TestStand Help</em>.</td>
</tr>
</tbody>
</table>
<TestStand Application Data>—Hidden by default and located at C:\ProgramData\National Instruments\TestStand on Windows 7/Vista and at C:\Documents and Settings\All Users\Application Data\National Instruments\TestStand on Windows XP.

<TestStand Local Application Data>—Hidden by default and located at <User Directory>\AppData\Local\National Instruments\TestStand on Windows 7/Vista and at <User Directory>\Local Settings\Application Data\National Instruments\TestStand on Windows XP.

**<TestStand> Directory**

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

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

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

Table 8-3. TestStand Component Subdirectories

<table>
<thead>
<tr>
<th>Directory Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyzer</td>
<td>Contains sequence analyzer built-in rules file, built-in analysis modules, and analysis module templates. Refer to the <em>NI TestStand Help</em> for more information about customizing sequence analyzer rules.</td>
</tr>
<tr>
<td>Callbacks</td>
<td>Contains the sequence files in which TestStand stores Station Engine callbacks and Front-End callbacks. Refer to Chapter 10, <em>Customizing Process Models and Callbacks</em>, for more information about customizing Station Engine and Front-End callbacks.</td>
</tr>
<tr>
<td>Compatibility</td>
<td>Contains type palette files TestStand uses to save sequence files compatible with earlier versions of TestStand.</td>
</tr>
<tr>
<td>Icons</td>
<td>Contains icon files for module adapters and step types.</td>
</tr>
<tr>
<td>Language</td>
<td>Contains string resource files in language-specific subdirectories. Refer to the <em>Creating String Resource Files</em> section of this chapter for more information about creating string resource files.</td>
</tr>
<tr>
<td>Models</td>
<td>Contains the default process model sequence files and supporting code modules. Refer to Chapter 10, <em>Customizing Process Models and Callbacks</em>, for more information about customizing process models.</td>
</tr>
<tr>
<td>Obsolete</td>
<td>Contains components TestStand no longer uses but installs to maintain backward compatibility.</td>
</tr>
<tr>
<td>Schemas</td>
<td>Contains the schema for XML-formatted TestStand sequence files.</td>
</tr>
<tr>
<td>StepTypes</td>
<td>Contains support files for step types. TestStand installs support files for the built-in step types in the <code>&lt;TestStand&gt;\Components\StepTypes</code> directory. Refer to Chapter 13, <em>Custom Step Types</em>, for more information about customizing step types.</td>
</tr>
<tr>
<td>Stylesheets</td>
<td>Contains the style sheet and supporting files to view sequence analyzer reports and TestStand File Diff/Merge application reports.</td>
</tr>
<tr>
<td>Tools</td>
<td>Contains sequences and supporting files for the Tools menu commands. Refer to the <em>Tools Menu</em> section of this chapter for more information about customizing the Tools menu.</td>
</tr>
</tbody>
</table>
Chapter 8 Customizing and Configuring TestStand

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 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 use the TestStand Deployment Utility to 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 the deployment utility.

### `<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>
<thead>
<tr>
<th>Directory Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdapterSupport</td>
<td>Support files for the LabVIEW and LabWindows/CVI Adapters.</td>
</tr>
<tr>
<td>CodeTemplates</td>
<td>Empty by default. You can store modified source code templates for step types, or code templates you create.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Directory Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>Placeholder subdirectories for modified TestStand components and components you develop, including callback files, language files, process model files, run-time servers, type palette files, and sequence analyzer analysis modules for custom rules you create. Refer to the <code>Components Directory</code> section of this chapter for more information about the subdirectories in the default <code>Components</code> directory TestStand installs.</td>
</tr>
<tr>
<td>Examples</td>
<td>Example sequences and tests. Most example sequences that use VIs call subVIs in the <code>&lt;LabVIEW&gt;\vi.lib</code> directory, which you can access after you install LabVIEW.</td>
</tr>
<tr>
<td>RuntimeServers</td>
<td>Contains example legacy source code for building a LabVIEW run-time application for executing LabVIEW-based code modules on a computer on which you have not installed the LabVIEW development system. Refer to the <code>Selecting a LabVIEW Server</code> section of Chapter 5, <code>Configuring the LabVIEW Adapter</code>, of the <code>Using LabVIEW and LabWindows/CVI with TestStand</code> manual for more information about using LabVIEW executables built with an ActiveX server enabled. Refer to Chapter 7, <code>Effectively Using LabVIEW with TestStand</code>, of the <code>Using LabVIEW and LabWindows/CVI with TestStand</code> manual for more information about how to best use LabVIEW features in a TestStand system.</td>
</tr>
<tr>
<td>Setup</td>
<td>Support files for the TestStand installer.</td>
</tr>
<tr>
<td>Tutorial</td>
<td>Sequences and code modules you use in tutorials in this manual, the <code>Using TestStand</code> manual, the <code>Using LabVIEW and LabWindows/CVI with TestStand</code> manual, and the <code>NI TestStand Evaluation Guide</code>. This directory also contains a <code>Solution</code> subdirectory that contains completed versions of the tutorial files.</td>
</tr>
<tr>
<td>UserInterfaces</td>
<td>Copies of the LabVIEW, LabWindows/CVI, Microsoft Visual Basic, C#, and C++ (MFC) user interfaces with source code installed in the <code>&lt;TestStand&gt;\UserInterfaces</code> directory.</td>
</tr>
</tbody>
</table>
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 installs the `<TestStand Application Data>` directory. This directory is hidden by default and contains configuration files users generally do not edit but an application can edit. The directory includes the `<Cfg>` subdirectory, which contains configuration files for TestStand Engine, TestStand Sequence Editor, and TestStand User Interface options, as shown in Table 8-5. You might need to deploy these files to a target computer when creating a deployment. Refer to Chapter 14, *Deploying TestStand Systems*, for more information about deployment.

Table 8-5. `<Cfg>` Subdirectory

<table>
<thead>
<tr>
<th>Directory or File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>SeqEdit directory</td>
<td>Default and saved layout files for the TestStand Sequence Editor.</td>
</tr>
<tr>
<td>ConnectionSource.ini</td>
<td>Datalink list the Database and Property Loader step types use for configuration.</td>
</tr>
<tr>
<td>Deployment Utility RecentFiles.ini</td>
<td>Most recently used file list for the deployment utility.</td>
</tr>
<tr>
<td>DeploymentUtilityOptions.ini</td>
<td>Configuration options for the deployment utility.</td>
</tr>
<tr>
<td>DocGen.ini</td>
<td>Configuration options for the Sequence File Documentation tool.</td>
</tr>
</tbody>
</table>
Table 8-5. <Cfg> Subdirectory (Continued)

<table>
<thead>
<tr>
<th>Directory or File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>EngineParts.ini</td>
<td>Deployment utility uses this file when you include the TestStand Engine in an installer. Note: Do not edit this file.</td>
</tr>
<tr>
<td>ExpressMenu</td>
<td>Cache of Express VI menu for the specified LabVIEW version.</td>
</tr>
<tr>
<td>Filter.ini</td>
<td>List of files to exclude from deployment.</td>
</tr>
<tr>
<td>PLPropertyLoader.ini</td>
<td>Configuration options for the Import/Export tool.</td>
</tr>
<tr>
<td>StationGlobals.ini</td>
<td>Definitions and values of station global variables.</td>
</tr>
<tr>
<td>Templates.ini</td>
<td>Templates configuration for the Insertion Palette in the sequence editors.</td>
</tr>
<tr>
<td>TestExec.ini</td>
<td>Configuration options for search directories, stations, adapters, the Insertion Palette menu, the file path history, and the type palette list.</td>
</tr>
<tr>
<td>TestStandModelOptions.ini</td>
<td>Model options for built-in process models.</td>
</tr>
<tr>
<td>TestStandModelReportOptions.ini</td>
<td>Report options for built-in process models.</td>
</tr>
<tr>
<td>TestStandDatabaseOptions.ini</td>
<td>Database options for built-in process models.</td>
</tr>
<tr>
<td>TestStandPersistedOptions.opt</td>
<td>Persisted breakpoints and watch expressions to use when no workspace is loaded.</td>
</tr>
<tr>
<td>ToolMenu.ini</td>
<td>List of Tools menu items.</td>
</tr>
<tr>
<td>Users.ini</td>
<td>List of TestStand users.</td>
</tr>
</tbody>
</table>

<TestStand Local Application Data> Directory

TestStand installs the <TestStand Local Application Data> directory. This directory is hidden by default and contains configuration files users generally do not edit but an application can edit, as shown in Table 8-6.
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>\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. When 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.

<table>
<thead>
<tr>
<th>File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout_current.bin</td>
<td>Layout of the TestStand Sequence Editor the last time it was closed</td>
</tr>
<tr>
<td>SeqEdit.xml</td>
<td>Configuration options for the TestStand Sequence Editor</td>
</tr>
<tr>
<td>UserInterface.xml</td>
<td>Configuration options for the user interfaces</td>
</tr>
</tbody>
</table>

---

**Table 8-6. `<TestStand Local Application Data>` Directory**
String Resource File Format

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

```
[categoy1]
tag1 = "string value 1"
tag2 = "string value 2"
[categoy2]
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:

```
[categoy1]
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 8-7 to insert unprintable characters.

<table>
<thead>
<tr>
<th>Escape Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Embedded linefeed character.</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return character.</td>
</tr>
<tr>
<td>\t</td>
<td>Tab character.</td>
</tr>
<tr>
<td>\xnn</td>
<td>Hexadecimal value that represents the character. For example, \x1B represents the ASCII ESC character.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Backslash character.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quotation mark.</td>
</tr>
</tbody>
</table>
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 8-8 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>
<thead>
<tr>
<th>Option</th>
<th>Purpose</th>
</tr>
</thead>
</table>
| `sequencefile` *(sequencefile2)*... | 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" |
| `/run sequence sequencefile`   | 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" |
| `/runEntryPoint entrypointname sequence file` | 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" |
| `/editor`                      | Instructs the application to open in Editor Mode when the application supports editing, as shown in the following example:  
    testexec.exe /editor |
| `/operatorInterface`           | Instructs the application to open in Operator Mode, as shown in the following example:  
    testexec.exe /operatorInterface |
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. When an error exists in the command-line argument, the Application Manager control generates an `ApplicationMgr.ReportError` event.

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>/quit</td>
<td>Instructs the application to exit after running the executions you specify, as shown in the following example:</td>
</tr>
<tr>
<td></td>
<td><code>SeqEdit.exe /run MainSequence &quot;c:\My Seqs\test\seq&quot; /quit</code></td>
</tr>
<tr>
<td></td>
<td>TestStand ignores the <code>/quit</code> option if the execution fails to launch.</td>
</tr>
<tr>
<td>/useExisting</td>
<td>Instructs the application to use the existing running instance of the application instead of opening a new instance, as shown in the following example:</td>
</tr>
<tr>
<td></td>
<td><code>SeqEdit.exe /useExisting</code></td>
</tr>
<tr>
<td></td>
<td>TestStand ignores the <code>/useExisting</code> option when you specify the <code>/quit</code> option.</td>
</tr>
<tr>
<td>/setCurrentDir</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td><code>SeqEdit.exe /setCurrentDir</code></td>
</tr>
<tr>
<td></td>
<td>The current directory is the directory the File dialog box initially displays when you open or save a file. Use this option to instruct the File dialog box to display 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.</td>
</tr>
<tr>
<td>/?</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td><code>SeqEdit.exe /?</code></td>
</tr>
<tr>
<td></td>
<td>TestStand ignores all other options when you specify the &quot;/?&quot; option.</td>
</tr>
</tbody>
</table>
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, and Chapter 15, Creating Custom User Interfaces in LabWindows/CVI, of the Using LabVIEW and LabWindows/CVI with TestStand manual
  When 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.
- Chapter 9, Using the TestStand ActiveX APIs in LabVIEW, and Chapter 16, Using the TestStand ActiveX APIs in LabWindows/CVI, of the Using LabVIEW and LabWindows/CVI with TestStand manual.

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. 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.

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

You can call 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.
- When you call the `Engine.NewExecution` method to create an execution, the TestStand UI Controls recognize the new execution.
- When 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.
• When you hold references to TestStand objects, release them in the handler for the `ApplicationMgr.QueryShutdown` event when the event handler does not cancel the shut down process.

• The TestStand UI Controls display only the active thread and do not support enabling the `ExecMask_TraceAllThreads` ExecutionMask constant.

• Do not couple user interfaces to specific sequences.

• Do not use instrumentation code in user interfaces.

• Use UI Messages to communicate information between code modules and user interfaces.

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.

• Loads type palette files at initialization.
• Logs users in and out.
• Loads and unloads sequence 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 Microsoft 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

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>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.</td>
</tr>
<tr>
<td>CheckBox</td>
<td>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.”</td>
</tr>
<tr>
<td>ComboBox</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table 9-1. Visible TestStand UI Controls (Continued)

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExpressionEdit</td>
<td>Use an ExpressionEdit control so users can edit a TestStand expression with syntax coloring, popup help, and statement completion. 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. 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.</td>
</tr>
<tr>
<td>InsertionPalette</td>
<td>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.</td>
</tr>
<tr>
<td>Label</td>
<td>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 unit under test (UUT).</td>
</tr>
<tr>
<td>ListBar</td>
<td>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.</td>
</tr>
<tr>
<td>ListBox</td>
<td>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.</td>
</tr>
<tr>
<td>ReportView</td>
<td>Connect an ExecutionView Manager control to a ReportView control to display the report for the selected execution.</td>
</tr>
<tr>
<td>SequenceView</td>
<td>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.</td>
</tr>
</tbody>
</table>
Connecting Manager Controls to Visible Controls

Connect a Manager control to visible controls 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.

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StatusBar</td>
<td>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.</td>
</tr>
<tr>
<td>VariablesView</td>
<td>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.</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>List</th>
<th>Manager Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapters</td>
<td>Application Manager</td>
</tr>
<tr>
<td>Sequence files</td>
<td><code>SequenceFileView Manager</code></td>
</tr>
<tr>
<td>Sequences</td>
<td><code>SequenceFileView Manager</code></td>
</tr>
<tr>
<td>Step groups</td>
<td><code>SequenceFileView Manager</code></td>
</tr>
<tr>
<td>Executions</td>
<td><code>ExecutionView Manager</code></td>
</tr>
<tr>
<td>Threads</td>
<td><code>ExecutionView Manager</code></td>
</tr>
<tr>
<td>Stack frames</td>
<td><code>ExecutionView Manager</code></td>
</tr>
</tbody>
</table>

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, when 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. When 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 whether 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. When no license exists, the ApplicationMgr.Start method returns an error the application displays before exiting. When an unactivated license or an evaluation license exists, the ApplicationMgr.Start method prompts users to activate a license.
When the `ApplicationMgr.IsEditor` property is True, the `ApplicationMgr.Start` method requires a license that permits editing. When 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 when it cannot obtain a license that permits editing.

**Note** You can call the `Engine.AcquireLicense` method before calling the `ApplicationMgr.Start` method to control whether TestStand launches a dialog box for evaluating, activating, or purchasing TestStand when the license is not properly activated.

### 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 and LabWindows/CVI 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`
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 15, *Creating Custom User Interfaces in LabWindows/CVI*, of the *Using LabVIEW and 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 a supported version of Visual Studio, select **Project»<Project Name>»Properties**, click the **Build** tab, and select **x86** from the **Platform Target** ring control in the General section of the 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»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* 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 `I` 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 the 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 Microsoft Foundation Class (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** When 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 when 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:
   ```cpp
   CWnd mExprEditCWnd;
   ```
2. Insert the following code into the OnInitDialog method of the dialog class:
   ```cpp
   mExprEditCWnd.Attach(GetDlgItem(IDC_MYEXPRESSIONEDIT)->m_hWnd);
   ```
3. Obtain the interface pointer from the CWnd member as follows:
   ```cpp
   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.
Creating Custom User Interfaces

Events

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.

Table 9-3. Creating Event Handlers in Specific ADEs

<table>
<thead>
<tr>
<th>ADE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LabVIEW</td>
<td>Register event handler VIs with the Register Event Callback function. Refer to the Handling Events section of Chapter 6, Creating Custom User Interfaces in LabVIEW, of the Using LabVIEW and LabWindows/CVI with TestStand manual for more information about handling events from the TestStand UI Controls in LabVIEW.</td>
</tr>
<tr>
<td>LabWindows/CVI</td>
<td>Install ActiveX event callback functions by calling the TSUI_&lt;object class&gt;EventsRegOn&lt;event name&gt; functions in tsui.fp. Refer to the Handling Events section of Chapter 15, Creating Custom User Interfaces in LabWindows/CVI, of the Using LabVIEW and LabWindows/CVI with TestStand manual for more information about handling events from the TestStand UI Controls in LabWindows/CVI.</td>
</tr>
<tr>
<td>.NET</td>
<td>Create .NET control event handlers from the form designer.</td>
</tr>
<tr>
<td>C++ (MFC)</td>
<td>Create ActiveX event handlers from the Message Maps page in the Class Wizard dialog box.</td>
</tr>
</tbody>
</table>
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. When 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. When the method
   returns True, exit the application. When 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. When 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 application development environments (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.
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.

<table>
<thead>
<tr>
<th>ADE</th>
<th>Help Location</th>
<th>Files</th>
<th>How to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>LabVIEW</td>
<td>Context help for each VI and in the \textit{NI TestStand VIs and Functions Help}, accessible by right-clicking the VI and selecting \textit{Help} from the shortcut menu or by selecting \textit{Help}\textgreater\NI TestStand VIs and Functions</td>
<td>VIs on the \textbf{Functions}\textgreater\textbf{TestStand} palette _TSUtility.lib located in &lt;TestStand&gt;\API\LabVIEW</td>
<td>Place VIs on the block diagram. Refer to Chapter 6, \textit{Creating Custom User Interfaces in LabVIEW}, of the \textit{Using LabVIEW and LabWindows/CVI with TestStand} manual for more information about using the TSUtil library in LabVIEW.</td>
</tr>
<tr>
<td>LabWindows/CVI</td>
<td>Function panels (TSUtil.fp)</td>
<td>TSUtil.c, TSUtil.h, TSUtil.fp, and TSUtil.obj located in &lt;TestStand&gt;\API\CVI</td>
<td>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 15, \textit{Creating Custom User Interfaces in LabWindows/CVI}, of the \textit{Using LabVIEW and LabWindows/CVI with TestStand} manual for more information about using the TSUtil library in LabWindows/CVI.</td>
</tr>
<tr>
<td>.NET Languages</td>
<td>In the Object Browser and in the source window using Intellisense</td>
<td>National Instruments. TestStand.Utility.dll located in &lt;TestStand&gt;\API\DotNet\Assemblies\CurrentVersion</td>
<td>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 \textit{Adding Assembly References in Visual Studio} section of this chapter for more information about adding references to assembly files in Visual Studio.</td>
</tr>
<tr>
<td>C++ (MFC)</td>
<td>Comments in the C++ header file, TSUtilCPP.h</td>
<td>TSUtilCPP.cpp and TSUtilCPP.h located in &lt;TestStand&gt;\API\VC</td>
<td>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.</td>
</tr>
</tbody>
</table>
Adding Assembly References in Visual Studio

Complete the following steps to add an assembly reference in Visual Studio.

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.

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 but do not support using the MenuStrip control in Visual Studio. To access the .NET MainMenu control in the Visual Studio 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

<table>
<thead>
<tr>
<th>ADE</th>
<th>Menu Open Notification Method</th>
</tr>
</thead>
</table>
| LabVIEW            | <This VI>:Menu Activation? event  
Refer to the Menu Bars and Menu Event Handling section of Chapter 6, Creating Custom User Interfaces in LabVIEW, of the Using LabVIEW and LabWindows/CVI with TestStand manual for more information about determining when a menu is about to open in LabVIEW. |
| LabWindows/CVI     | InstallMenuDimmerCallback  
Refer to the Menu Bars section of Chapter 15, Creating Custom User Interfaces in LabWindows/CVI, of the Using LabVIEW and LabWindows/CVI with TestStand manual for more information about determining when a menu is about to open in LabWindows/CVI. |
| .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.
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 whether 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 whether 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.

When a Sequence File window attempts to close and the SequenceFileViewMgr.SequenceFile property is NULL, the application closes the window immediately. When 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 whether 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.

When an Execution window attempts to close and the ExecutionViewMgr.Execution property is NULL, the application closes the window immediately. When 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, Insertion Palette step type and template list settings, and other settings that apply to all TestStand applications in files such as TestExec.ini and Templates.ini, located in the <TestStand Application Data>\Cfg directory. 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 when a configuration file does not already exist. When the configuration file already exists, the Application Manager control loads the values of these properties from the file.

Refer to the <TestStand Application Data> Directory section of Chapter 8, Customizing and Configuring TestStand, for more information about the files in the <TestStand Application Data>\Cfg directory.

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>\AppData\Local\National Instruments\TestStand on Windows 7/Vista and <User Directory>\Local Settings\Application Data\National Instruments\TestStand on Windows XP. Set the ApplicationMgr.ConfigFilePath property before the application calls the ApplicationMgr.Start method to change the configuration file location.
When you specify a relative file path or just a filename, the file location is relative to the directory that contains the application. When 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.

Refer to the `<TestStand Application Data> Directory` section of Chapter 8, *Customizing and Configuring TestStand*, for more information about the configuration files in the `<TestStand Application Data>\Cfg` directory.

### 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 NI TestStand User Interfaces* document, located at `<TestStand>\Doc\UsingtheTestStandUserInterfaces.html`, 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

You can use the TestStand Deployment Utility to deploy a user interface. Refer to Chapter 14, *Deploying TestStand Systems*, for more information about the deployment utility.

Authenticode Signatures for Windows

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. In addition, when an application launches on Windows 7/Vista, User Account Control (UAC) determines whether to grant the application administrator privileges and displays your company as the publisher when confirming elevation. Refer to Microsoft documentation for more information about Authenticode signatures.

Add an Authenticode signature to a TestStand user interface you create when 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 when the user interface requires administrator privileges to run on Windows 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 when 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](http://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. 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, when you run the sequence editor or example user interfaces as administrator on
Windows, 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 7/Vista, the UAC security component determines whether to grant the application administrative privileges. A user that logs into Windows as a standard user can write only to specific locations on disk and in the registry. Standard user is the default login.

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.

When an application does not specify an execution level in a 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 or later automatically include a default manifest in built applications, and with LabVIEW 2009 or later, you can specify a custom manifest for built applications. With LabWindows/CVI 8.5 or later, you can specify a manifest for built applications. When you build the application, refer to the documentation for the ADE you used for more information about how to include a manifest.
Customizing Process Models and Callbacks

You can customize the default process models and callbacks. Customize callbacks to implement custom functionality specific to unit under test (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 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 or to create a new process model, copy all the process model files from the <TestStand>\Components\Models\TestStandModels directory to the <TestStand Public>\Components\Models 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.
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 the sequence as 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.
4. Select Callback from the Type ring control.
5. Click **OK**.

6. 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.

**Note** You can also override a callback in the process model sequence file by creating a sequence with the same name but different functionality in StationCallbacks.seq in the `<TestStand Public>\Components\Callbacks\Station` directory. TestStand calls model callback sequences in StationCallbacks.seq instead of calling the similarly named callback defined in any model. A callback sequence in a client file overrides the similarly named callback in the model and the StationCallbacks.seq 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. Each entry point is a sequence in the process model file.

Execution entry points run test programs typically by calling the MainSequence callback in the client sequence file. The 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 in the `<TestStand Application Data>\Cfg` directory. The process models contain the following Configuration entry points:

- **Configure 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.
- **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.
- **Configure 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 \Components\Callbacks\Station directory.

**Note** TestStand does not predefine any Station Engine callbacks in StationCallbacks.seq in the \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>
<thead>
<tr>
<th>Engine Callback</th>
<th>Where You Define the Callback</th>
<th>When the Engine Calls the Callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>SequenceFilePreStep</td>
<td>Any sequence file</td>
<td>Before the engine executes each step in the sequence file.</td>
</tr>
<tr>
<td>SequenceFilePostStep</td>
<td>Any sequence file</td>
<td>After the engine executes each step in the sequence file. The engine does not call this callback when a step errors.</td>
</tr>
<tr>
<td>SequenceFilePreInteractive</td>
<td>Any sequence file</td>
<td>Before the engine begins an interactive execution of steps in the sequence file.</td>
</tr>
<tr>
<td>SequenceFilePostInteractive</td>
<td>Any sequence file</td>
<td>After the engine completes an interactive execution of steps in the sequence file.</td>
</tr>
<tr>
<td>SequenceFileLoad</td>
<td>Any sequence file</td>
<td>When the engine loads the sequence file into memory.</td>
</tr>
</tbody>
</table>
Table 10-1. Engine Callbacks (Continued)

<table>
<thead>
<tr>
<th>Engine Callback</th>
<th>Where You Define the Callback</th>
<th>When the Engine Calls the Callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>SequenceFileUnload</td>
<td>Any sequence file</td>
<td>When the engine unloads the sequence file from memory.</td>
</tr>
<tr>
<td>SequenceFilePostResultListEntry</td>
<td>Any sequence file</td>
<td>After the engine fills out the step result for a step in the sequence file.</td>
</tr>
<tr>
<td>SequenceFilePostStepRuntimeError</td>
<td>Any sequence file</td>
<td>After a step in the sequence file generates a run-time error.</td>
</tr>
<tr>
<td>SequenceFilePostStepFailure</td>
<td>Any sequence file</td>
<td>After a step in the sequence file fails.</td>
</tr>
<tr>
<td>ProcessModelPreStep</td>
<td>Process model file</td>
<td>Before the engine executes each step in any client sequence file the process model calls and each step in any resulting subsequence calls.</td>
</tr>
<tr>
<td>ProcessModelPostStep</td>
<td>Process model file</td>
<td>After the engine executes each step in any client sequence file the process model calls and each step in any resulting subsequence calls. The engine does not call this callback when a step errors.</td>
</tr>
<tr>
<td>ProcessModelPreInteractive</td>
<td>Process model file</td>
<td>Before the engine begins an interactive execution of steps in a client sequence file and steps in any resulting subsequence calls.</td>
</tr>
<tr>
<td>Engine Callback</td>
<td>Where You Define the Callback</td>
<td>When the Engine Calls the Callback</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ProcessModelPostInteractive</td>
<td>Process model file</td>
<td>After the engine completes an interactive execution of steps in a client sequence file and steps in any resulting subsequence calls.</td>
</tr>
<tr>
<td>ProcessModelPostResultListEntry</td>
<td>Process model file</td>
<td>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.</td>
</tr>
<tr>
<td>ProcessModelPostStepRuntimeError</td>
<td>Process model file</td>
<td>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.</td>
</tr>
<tr>
<td>ProcessModelPostStepFailure</td>
<td>Process model file</td>
<td>After a step fails when the step is in a client sequence file the process model calls or in any resulting subsequence calls.</td>
</tr>
<tr>
<td>StationPreStep</td>
<td>StationCallbacks.seq</td>
<td>Before the engine executes each step in any sequence file.</td>
</tr>
<tr>
<td>StationPostStep</td>
<td>StationCallbacks.seq</td>
<td>After the engine executes each step in any sequence file. The engine does not call this callback when a step errors.</td>
</tr>
<tr>
<td>StationPreInteractive</td>
<td>StationCallbacks.seq</td>
<td>Before the engine begins any interactive execution.</td>
</tr>
<tr>
<td>StationPostInteractive</td>
<td>StationCallbacks.seq</td>
<td>After the engine completes any interactive execution.</td>
</tr>
</tbody>
</table>
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, when 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:

- When 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.

- When a callback sequence is empty, TestStand does not invoke the Engine callback.

- Process models use the Execution.EnableCallback method to disable the ProcessModelPostResultListEntry callback when the model does not need to process results on-the-fly for report generation or database logging.

<table>
<thead>
<tr>
<th>Engine Callback</th>
<th>Where You Define the Callback</th>
<th>When the Engine Calls the Callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>StationPostResultListEntry</td>
<td>StationCallbacks.seq</td>
<td>After the engine fills out the step result for a step in any sequence file.</td>
</tr>
<tr>
<td>StationPostStepRuntimeError</td>
<td>StationCallbacks.seq</td>
<td>After any step generates a run-time error.</td>
</tr>
<tr>
<td>StationPostStepFailure</td>
<td>StationCallbacks.seq</td>
<td>After any step fails.</td>
</tr>
</tbody>
</table>

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, when 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:

- When 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.

- When a callback sequence is empty, TestStand does not invoke the Engine callback.

- Process models use the Execution.EnableCallback method to disable the ProcessModelPostResultListEntry callback 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.

When TestStand reloads a sequence file after the file is modified on disk, TestStand can call the SequenceFileLoad callback in the new file without calling the SequenceFileUnload callback in the original file.

**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 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 when 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. When 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 TestStand Sequence Editor.
When you modify a type, TestStand enables the `PropTypeFlags_IsModifiedType` type flag for the type. TestStand cannot automatically resolve type conflicts unless you disable the `PropTypeFlags_IsModifiedType` type flag. To disable the `PropTypeFlags_IsModifiedType` type flag, 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 File tab of the Station Options dialog box is set to Prompt to Increment Type Versions. This causes TestStand to launch the Modified Types Warning dialog box when you select **File»Save <filename>** and the sequence file or type palette contains types that are marked as modified. You can increment the type version or remove the modified mark on the type before saving, or save the type as modified.

When you open a file that was saved in an earlier version of TestStand, TestStand marks the file as modified when it automatically converts the file and upgrades built-in types and custom types that the file uses. Refer to the *NI TestStand Help* for more information about using the Do Not Mark Files as Modified if Modification is Caused Only by Automatic TestStand Version Upgrade option on the File tab of the Station Options dialog box to configure whether TestStand marks files as modified when upgrading files.

### Type Versioning

You can use only one version of a type at a time. TestStand uses the version number of a type to determine which version of a type to use when opening a file that has a different version of the type than the version currently in memory. Typically, the version of the type with the latest version number is the version that TestStand automatically uses.

In versions of TestStand earlier than version 4.1, the propagation of unwanted type versions between files could occur when you open a sequence file with a version of a type later than the version of the type in a type palette file. The automatic type conflict resolution in TestStand updated the type palette file and every file you subsequently opened to use the later version without warning or notification.

In TestStand 4.1 or later, if the type is in a type palette file, the latest version must also be the version in the type palette file and the earlier version must be in a file other than a type palette file for automatic type conflict resolution to occur. This behavior avoids unwanted type propagation.
can modify the behavior to be more or less strict by changing the value of the Allow Automatic Type Conflict Resolution option on the File tab of the Station Options dialog box.

Use the Set Earliest TestStand Versions that can Use this Type option on the Version tab of the Type Properties dialog box to specify the earliest TestStand version that can use a type to prevent the TestStand Engine from using the type when the version of the engine is earlier than the TestStand version you specify. When 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 when an earlier version of the type already exists in memory. If you do not specify the earliest TestStand version that can use a type and you set the Allow Automatic Type Conflict Resolution option on the File tab of the Station Options dialog box to Always, future versions of a type might propagate into type palette files and sequence files that run in the previous TestStand version.

Refer to the Managing Type Revisions section of this chapter and to the NI TestStand Help for more information about the Allow Automatic Type Conflict Resolution option and avoiding unwanted type propagation.

Managing Type Revisions

When 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 latest version number when all of the following conditions exist:

- The version numbers of the types are different
- The PropTypeFlags_IsModifiedType type flag 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 File tab of the Station Options dialog box does not restrict automatic type conflict resolution in this situation
If TestStand cannot automatically resolve a type conflict, TestStand either launches the Type Conflict in File dialog box or, if TestStand is loading a file an execution calls, returns a run-time error.

**Note** If an execution calls a file that has a later version of a type than the type version currently loaded in memory, TestStand does not automatically resolve the conflict and instead returns a run-time error. Replacing a type definition while an execution is running is unsafe because the execution might be using the type.

The Type Conflict in File dialog box prompts you to resolve the conflict. You can select to use or rename the type currently loaded in memory or use or rename the type in the file you are attempting to load. When you select to use one of the two versions of the type, TestStand converts all instances of the type to match the type version you select. If you rename one of the two versions of the type, TestStand modifies the instances of the type in memory that refer to the version of the type you select to refer to the renamed version.

**Note** Type instances not in memory still refer to the original name for the type and might cause additional conflicts when you load them.

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.

### Avoiding Unwanted Type Version Propagation

When a type conflict occurs, selecting to use the later version number of a type can propagate unwanted type versions to all instances of the type. For example, unwanted propagation might occur in the following situations:

- You create a new version of a step type that is not compatible with previous versions of TestStand. A sequence file that contains the new type loads in a previous version of TestStand that contains an older version of the type in a type palette. In versions of TestStand earlier than version 4.1, the new version of the type automatically propagates to the type palette in the previous version of TestStand and to all files loaded in memory from then on. Because the type is not compatible with previous versions of TestStand, you might experience undesired behavior when you run sequences that use the new type.

In TestStand 4.1 or later, by default types no longer automatically propagate to type palette files.
• Two types with completely different functionality share the same name. When the TestStand Engine loads the two types, the engine assumes that the types are different versions of the same type and loads the type with the later version number, which can affect the functionality of the sequence files that contain instances of the second type.

• Two different developers independently customize the same type with different changes.

Avoiding Unwanted Type Version Propagation to Previous Versions of TestStand

Because you can save sequence files to previous versions of TestStand but some types might not run correctly on previous versions of TestStand, you can specify the earliest version of TestStand that can run a type. Enable the Set Earliest TestStand Version that can Use this Type option on the Version tab of the Step Type Properties dialog box or Type Properties dialog box and set the earliest version to the current version of TestStand to prevent the current version of a type from being used in or accidentally propagated to an earlier TestStand version.

You can also create different versions of the type to run in different versions of TestStand. When you save a sequence for a previous version of TestStand, TestStand searches a set of compatibility directories to find the type version that is compatible with the previous version for which you want to save the sequence file. TestStand saves the types from the type palette files in the <TestStand>\Components\Compatibility\<version number> and <TestStand Public>\Components\Compatibility\<version number> directories with the sequence file. You can 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.

Using Type Palette Files to Restrict Automatic Type Conflict Resolution

To avoid unwanted type version propagation, create all types in type palette files and use the default behavior of the Allow Automatic Type Conflict Resolution option on the File tab of the Station Options dialog box. The default value for this option is Only if Type Palette Files will not be Modified (default) so that TestStand does not allow automatic type conflict resolution when the outcome of the resolution modifies a type palette file. This ensures that the application never uses a version of a type...
that is different than the version of the type in the type palette file without an explicit confirmation through the Type Conflict in File dialog box.

You can further restrict automatic conflict resolution by selecting **Only if a Type Palette File has the Higher Version** for the Allow Automatic Type Conflict Resolution option. This option includes the same restrictions as the Only if Type Palette Files will not be Modified (default) option and includes the restriction that the type must be in a type palette file for automatic type conflict resolution to occur. Effectively, this option allows automatic type conflict resolution only in the case in which a type palette file has the later version of the type and a non-type palette file has the earlier version of the type. For example, this option does not allow automatic type conflict resolution between two sequence files for types that are not in type palette files. The Only if Type Palette Files will not be Modified (default) option does allow this.

If you specify **Always (same as TestStand 4.0.x or earlier)** for the Allow Automatic Type Conflict Resolution option, TestStand always automatically resolves type conflicts by using version numbers, which can cause unintended propagation of types between files. For example, if you open a sequence file with a version of a type later than the version of the type currently in a type palette file, TestStand updates the type palette file and every file you subsequently open to use the later version. The updated type propagates to other files without warning or notifying you.

If you specify **Never** for the Allow Automatic Type Conflict Resolution option, TestStand does not automatically resolve type conflicts under any circumstances. When TestStand loads two different versions of a type, TestStand always prompts you or reports a type conflict error. When you select this option, opening files from TestStand versions earlier than the current version almost always results in type conflict prompts. Use this option only for debugging purposes or to ensure that all files have exactly the same version of every type.

Table 11-1 lists each of the Allow Automatic Type Conflict Resolution options and common scenarios for using each option.
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. A yellow plus sign icon, shown at left, indicates that the type is 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 <filename> 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 when the TestStand 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 copying a type palette file to the `<TestStand Public>\Components\TypePalettes` directory. Right-click the View Types For pane in the Types window and select Customize Type Palettes from the context menu to launch the Configure Type Palettes dialog box, in which you can specify which type palette files TestStand uses.

TestStand also 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 on 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 to 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 user manager.
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

<table>
<thead>
<tr>
<th>Context Menu Item</th>
<th>Location of Context Menu</th>
<th>Item Inserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert File Global</td>
<td>File Globals section of the Variables pane in the Sequence File window</td>
<td>Sequence file global variable</td>
</tr>
<tr>
<td>Insert Parameter</td>
<td>Parameters section of the Variables pane in the Sequence File window</td>
<td>Sequence parameter</td>
</tr>
<tr>
<td>Insert Local</td>
<td>Locals section of the Variables pane in the Sequence File window</td>
<td>Sequence local variable</td>
</tr>
<tr>
<td>Insert Station Global</td>
<td>Station Globals window</td>
<td>Station global variable</td>
</tr>
<tr>
<td>Insert User</td>
<td>User Manager window</td>
<td>New object with the User data type</td>
</tr>
<tr>
<td>Insert Group</td>
<td>User Manager window</td>
<td>New object with the User data type</td>
</tr>
<tr>
<td>Insert Field</td>
<td>Types window</td>
<td>New element in an existing data type</td>
</tr>
</tbody>
</table>
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**.

When 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. When 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.
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. You can also right-click the variable, parameter, or property on the Types pane of the Types window, 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

Empty Arrays

Enable the Empty option in the Array Bounds dialog box or on the Bounds tab of the Type Properties dialog box when 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, when 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. When the variable, parameter, or property is an object reference, you can use the **Release Object** button, which displays only when 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 scope. For example, when 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.

When 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.
Chapter 12  Standard and Custom Data Types

Error and CommonResults

TestStand inserts a Results property in every step you create. The Results property includes at least three subproperties—Error, Status, and Common.

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 Common 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.

The default CommonResults standard data type version (3.1.0.100) is empty and does not change in newer versions of TestStand to ensure that any use of a newer version of CommonResults in sequence files is maintained and not automatically overridden in a newer version of TestStand.

Note  TestStand 4.1 or later includes code to handle an issue in which sequence files with an unsupported version of CommonResults (3.5.0.1) were made publicly available. If you load a file with an empty CommonResults 3.5.0.1 data type, TestStand does not automatically use that version of the data type and instead uses the existing global version. This behavior ensures that TestStand does not automatically propagate this unintentionally released version of the type to other files.

In addition, when TestStand attempts to load two versions of the CommonResults data type and one version of the data type is defined as an empty container and has a version of 3.1.0.100 or earlier, and the other version of the data type is defined as a non-empty container, TestStand does not allow the empty version to automatically override the non-empty version, even if the empty version has a later version number. This behavior ensures that TestStand does not automatically lose changes made to CommonResults in versions of TestStand earlier than TestStand 3.1.

When 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. You can avoid the type conflicts by resaving the type palette files and instructing TestStand to increment the type version for CommonResults before opening any other files. National Instruments recommends modifying the CommonResults data type only when 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 when 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. When 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, comments, and attributes for the property. Click the Advanced button and select Flags from the popup menu to launch the Edit Flags dialog box, in which you can modify the property flags, or select Attributes from the popup menu to launch the Attributes dialog box, in which you can modify the property attributes. 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 and Attributes dialog box.

- **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 whether 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. Refer to Chapter 7, *Effectively Using LabVIEW with TestStand*, of the *Using LabVIEW and LabWindows/CVI with TestStand* manual for more information about how to best use LabVIEW features in a TestStand system.

- **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.
Chapter 12  Standard and Custom Data Types

- **.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 to access the context menu, select **Insert Field**, 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 and select an adapter.

7. Click Add and select the type of step to create substeps. Use the Substep Info section on the Substeps tab to specify the menu item name of Edit steps.

**Note** Step Settings panes are not available for custom step types you create. You must use dialog boxes to set custom elements of the step types you create.

### 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 TestStand 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. When 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 when 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 to the *NI TestStand Help* for more information about the Disable Properties tab.

Click the **Advanced** button to launch a popup menu that contains the following options:

- **Flags**—Launches 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.

- **Attributes**—Launches the Attributes dialog box, in which you can create and edit the default attributes for the steps you create with the step type.

- **Block Structure**—Launches the Block Structure dialog box, in which you can modify the block structure settings for steps you create with the step type. The Flow Control step types, such as If, ElseIf, and End, use these built-in properties.

- **Default Additional Results**—Launches the Configure Default Additional Results dialog box, in which you can configure default
additional results for the step. The step type must have a default module to configure default parameter additional results.

- **Preconfigured Additional Results**—Launches the Configure Preconfigured Additional Results dialog box, in which you can configure the additional results hints for the step type. The additional results hints define a list of preconfigured custom additional results you can choose to log when you edit the additional results of a step in a user interface.

Refer to the *NI TestStand Help* for more information about the Edit Flags dialog box, Attributes dialog box, Block Structure dialog box, Configure Default Additional Results dialog box, and Configure Additional Results Hints 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, click 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. If the Edit substep modifies any step properties, increment the change count of the sequence file by calling the PropertyObjectFile.IncChangeCount method so an editing application can determine that the file changed and refresh itself.

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 Microsoft Foundation Class (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 file 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

<table>
<thead>
<tr>
<th>Subdirectory Name</th>
<th>Template Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default_Template</td>
<td>Legacy default template</td>
</tr>
<tr>
<td>DefaultC++.NET</td>
<td>Default template for C++ in Microsoft Visual Studio</td>
</tr>
<tr>
<td>DefaultCSharp.NET</td>
<td>Default template for C# in Visual Studio</td>
</tr>
<tr>
<td>DefaultCVI</td>
<td>Default template for C in LabWindows/CVI</td>
</tr>
<tr>
<td>DefaultHTB72_Template</td>
<td>Default template for HTBasic 7.2</td>
</tr>
<tr>
<td>DefaultHTB80_Template</td>
<td>Default template for HTBasic 8.0</td>
</tr>
<tr>
<td>DefaultLabVIEW</td>
<td>Default template for LabVIEW</td>
</tr>
<tr>
<td>DefaultVB.NET</td>
<td>Default template for Microsoft Visual Basic NET</td>
</tr>
<tr>
<td>DefaultVC++_Template</td>
<td>Default template for C++ in Visual Studio</td>
</tr>
</tbody>
</table>

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. When these two sources of information do not match, TestStand prompts you to select which prototype to use for the skeleton code. When 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 of 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 or 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 or 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 when 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 when it exists.

**Version Tab**

Use the Version tab to edit the version information for the data type, to determine whether 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, when 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.
14

Deploying TestStand Systems

Use the TestStand Deployment Utility to create a deployable image of a TestStand system and, optionally, an installer that contains the deployable image. The deployable image can contain sequence files, code modules and supporting files, process model and supporting files, user interface applications, configuration files, and step types and supporting files the TestStand system uses.

You can also use the deployment utility to include the TestStand Engine and supporting files, LabVIEW and LabWindows/CVI Run-Time Engines (RTEs), and hardware drivers in the installer you create. The installer the deployment utility creates can also register ActiveX Automation servers, replace existing files on the target computer, and create program shortcuts. You can configure the deployment utility to remove VI block diagrams or to lock the VIs you deploy.

Note The versions of the 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 RTEs
• 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.

Creating a Deployment

Complete the following tasks, as outlined in the following sections, to deploy a TestStand test system using the TestStand Deployment Utility.

- Identify the components to deploy
- Create a workspace file or use a directory to specify which files to deploy
- Configure and build the deployment
- Determine whether you need to create an installer for the system

Identifying Components to Deploy

You can deploy the following types of files:

- Components in the <TestStand Public> directory
- Sequence files
- Code modules, including .NET and ActiveX code modules and dynamically called code modules and support files
- Support files, such as property loader files, image files associated with Message Popup steps, or other data files
- Configuration files, such as TestExec.ini
- User interface files
- TestStand Engine
- Required RTEs
- National Instrument hardware drivers
- Third-party drivers and installers
- Documentation

Refer to the <TestStand Application Data> Directory section of Chapter 8, Customizing and Configuring TestStand, for more information about the configuration files in the <TestStand Application Data>\Cfg directory. Refer to the TestStand Directory Structure section of Chapter 8, Customizing and Configuring TestStand, for more information about other TestStand directories.
Organizing Files for Deployment

To deploy a system, you must use either a workspace file or a directory and its subdirectories to specify which files to deploy. The workspace file or directory must contain all the types of files you use, as listed in the Identifying Components to Deploy section of this chapter.

When 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 use a workspace file or directory for the deployment.

Refer to the Workspaces section of Chapter 2, Sequence Files and Workspaces, for more information about workspace files.

Configuring the Deployment

Select Tools»Deploy TestStand System in the TestStand Sequence Editor or in a TestStand 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. You can also select Start»All Programs»National Instruments»TestStand»Tools»Deploy TestStand System to launch the deployment utility. Refer to the NI TestStand Help for more information about the TestStand Deployment Utility.

Determining Whether to Create an Installer

When 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 when you plan to use a third-party installer development tool, such as Wise or InstallShield, or when you plan to use a source code or revision control system to deploy the system files to target computers. In these cases, use the deployment utility to create a deployable image for the third-party installer tools or source code or revision control systems. The Location of Deployable Image option on the System Source Tab of the TestStand Deployment Utility determines the location of the deployable image.
Deployment Utility Build Process

The TestStand Deployment Utility collects and filters files to include in the deployment, processes support files 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. Some dependencies are not required for a deployed system to function correctly and might affect the performance of the deployed system.

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.

Filtering Files

Because distributing every file sequences use might be problematic, the deployment utility includes a filtering function that removes potentially unwanted files. For example, when steps in a sequence call functions in Microsoft 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 7/Vista to a target computer running Windows XP might result in DLL version incompatibility issues.
The Filter.ini file contains different sections for filtering files based on certain criteria. Each section contains filters, which are regular expressions that use the Match Pattern Function VI syntax.

Use the following examples to help you define a filter:

- To exclude components from a product of a certain company, use the [Exclude_Company_Product] section. Filters in this section take the following syntax:
  
  COMPANY_#="Name of company"
  PRODUCT_#="Name of product"

  During deployment, the deployment utility checks the version information strings for company and product names for all included files and excludes the matching files.

  For example, to exclude all TestStand and LabWindows/CVI files, use the following filter:

  [Exclude_Company_Products]
  COMPANY_0="National Instruments"
  PRODUCT_0="TestStand"
  COMPANY_1="National Instruments"
  PRODUCT_1="LabWindows.CVI"

- To exclude all files that have a certain filename, use the [Excluded_Filenames] section. Filters in this section take the following syntax:

  FILE_#="Name of the file to filter, including file extension"

  For example, to exclude all files named readme.txt or image.jpg, use the following filter:

  [Excluded_Filenames]
  FILE_0="readme.txt"
  FILE_1="image.jpg"

  Note The [Excluded_Filenames] section does not support regular expressions.

- To exclude all files from a certain directory, use the [Excluded_Dirs] section. Filters in this section take the following syntax:

  DIR_#="Name of directory"

  For example, to exclude all files from the Program Files directory and files from the Windows public directory, use the following filter:
[Excluded_Dirs]
DIR_0="Program Files"
DIR_1="Users\Public"

Note You do not have to specify a full directory path. The filter excludes any directory whose name matches the specified regular expression.

You can also use the predefined [Windows], [System32], [TestStand], and [TestStandPublic] sections to exclude files from those directories and their subdirectories. You can also create a custom section whose name is the absolute path of a directory to exclude files from the directory. Enter a regular expression for the filter to specify the files you want to exclude. Filters in these sections take the following syntax:
FILTER_#="Regular expression"

For example, to exclude all executable files from the Windows and Windows\System32 directories and their subdirectories, use the following filters:
[Windows]
FILTER_0=".*exe"
[System32]
FILTER_0=".*exe"

To exclude all files from the <TestStand>\Bin directory, use the following filter:
[TestStand]
FILTER_0="BIN\.*"

To exclude all .txt and .jpg files from the C:\Users folder and its subdirectories, use the following filter:
[C:\Users]
FILTER_0=".*txt"
FILTER_1=".*jpg"

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 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 have only the LabVIEW
RTE installed. When 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 section of Chapter 7, Effectively Using LabVIEW with TestStand, of the Using LabVIEW and LabWindows/CVI with TestStand manual for more information about restrictions for deploying LabVIEW VIs.

Refer to the NI TestStand Help for information about options you can set in the LabVIEW VI Options dialog box to choose between faster performance and improved error handling.

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 when 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

When 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 RTEs, 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 DVD that ships with TestStand or from a later version of the driver DVD. 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. 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 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 when 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 when you edit any deployed system files because the deployed system might not function properly otherwise.

- Install the complete drivers from the NI Device Driver DVD 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.

- Manually add required configuration files to the workspace. TestStand uses the default configuration for the deployment unless you include
modified configuration files. Some of the common configuration options to include are search directories, adapter configuration, station globals, report options, database options, and so on. Refer to the Identifying Components to Deploy section of this chapter for more information about files to include.

- Always ensure that you have a user interface or custom sequence editor to execute TestStand sequence files on the computer where you deploy the test system. Refer to the Distributing a User Interface section of this chapter for more information about deploying user interfaces.

- When you deploy LabVIEW VIs, make sure the version of the LabVIEW RTE you include with the deployment matches the version of LabVIEW you use to build the deployment. Additionally, if no LabVIEW development system is detected on the computer, the LabVIEW Adapter is configured by default to use Autodetect using VI version for the LabVIEW Run-Time Engine option in the LabVIEW Adapter Configuration dialog box to automatically use the LabVIEW RTE version that corresponds to the VI to run the VI.

Refer to the Building a TestStand Deployment with LabVIEW section of Chapter 7, Effectively Using LabVIEW with TestStand, of the Using LabVIEW and LabWindows/CVI with TestStand manual for more information about restrictions for deploying LabVIEW 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 Licensing Options for TestStand Systems section of Chapter 1, Introduction to TestStand, of the Using TestStand manual and 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»From TestStand Public 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, user interfaces, and language strings, you saved in the <TestStand Public> subdirectories.

3. On the Installer Options tab, use the Installer Directory field to specify the directory to which the deployment utility copies the installer for the system.

4. On the Installer Options tab, enable the Install TestStand Engine option.

5. 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.

6. Expand the TestStand Development Components section and enable the TestStand Sequence Editor option to include the application in the engine installation.

7. 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.

8. Click Save and save the build as EngineInstaller.tsd.

9. Click the Build button to create the installer.

10. To use the installer, copy all the files from the installer directory you specified on the Installer Options tab to a DVD or CD or to a shared directory on a network.

11. On the target computer, insert the DVD or CD or connect to the network and run the setup.exe application to start the installer.

12. When the installation completes, select Start»All Programs»National Instruments»TestStand»Sequence Editor to verify that the TestStand Engine installed correctly.

13. Activate a license when 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.

3. On the **Installer Options** tab, use the **Installer Directory** field to specify the directory to which the deployment utility copies the installer for the system.

4. Click the **File Browse** button located next to the From TestStand Workspace File control.

5. Browse to the `<TestStand Public>\Examples\Deployment` directory and select the `test.tsw` workspace file.

6. Click **Open**.

7. Click the **Distributed Files** tab.

8. Click **Yes** in the dialog box that launches to allow the deployment utility to analyze the workspace file and dependent files.

9. Click the **Build Status** tab and review the Status Log to check for analysis result warnings.

10. Disable the **unused.dll** option to remove it from the distribution because the example test system does not use this DLL.

11. On the **Installer Options** tab, enable the **Install TestStand Engine** option.

12. 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.

13. Expand the **TestStand Development Components** section and enable the **TestStand Sequence Editor** option to include the application in the engine installation.

14. 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.

15. Click **Save** to save the build as `test.tsd`.

16. Click the **Build** button to create the installer.
17. To use the installer, copy all the files from the installer directory you specified on the Installer Options tab to a DVD or CD or to a shared directory on a network.

18. On the target computer, insert the DVD or 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»National Instruments»TestStand»Sequence Editor to verify that the TestStand Engine installed correctly.

20. Activate a license when the sequence editor prompts you to do so.

21. 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. To deploy a dynamically called file, you must add the file to a workspace that you will deploy.

3. On the Installer Options tab, use the Installer Directory field to specify the directory to which the deployment utility copies the installer for the system.

4. Click the File Browse button located next to the From TestStand Workspace File control.

5. Browse to the <TestStand Public>\Examples\Deployment directory and select the Dynamically_called_sequence.tsw workspace file.

6. Click Open.

7. Click the Distributed Files tab.

8. Click Yes in the dialog box that launches to allow the deployment utility to analyze the workspace file and dependent files.

9. 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.

10. Click the Distributed Files tab and notice that Dynamic.seq is missing in the Distributed Files list.
11. In the sequence editor, load the `<TestStand Public>\Examples\Deployment\Dynamically_called_sequence.tsw` workspace file.

12. Add `<TestStand Public>\Examples\Deployment\Dynamic.seq` to the workspace file and save the changes.

13. 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`.

14. 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.

15. On the **Installer Options** tab, enable the **Install TestStand Engine** option.

16. Click the **Engine Options** button to launch the TestStand Engine Options dialog box, in which you select the components to include in the installer.

17. Expand the **TestStand Development Components** section and enable the **TestStand Sequence Editor** option to include the application in the engine installation.

18. 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.

19. Click **Save** to save the build as `Dynamic.tsd`.

20. Click the **Build** button to create the installer.

21. To use the installer, copy all the files from the installer directory you specified on the Installer Options tab to a DVD or CD or to a shared directory on a network.

22. On the target computer, insert the DVD or CD or connect to the network and run the `setup.exe` application to start the installer.

23. When the installation completes, select **Start»All Programs»National Instruments»TestStand»Sequence Editor** to verify that the TestStand Engine installed correctly.

24. Activate a license when the sequence editor prompts you to do so.

25. 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 RTE version that corresponds to the development environment version you use to create user interfaces. Refer to the Installing Additional Software Components section of the NI TestStand Release Notes for information about the RTE 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.
3. Save the project as User Interface.tpj.
4. Right-click the User Interface project and select Add Files to Project from the context menu.
5. 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 (*.*)
6. 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.
7. Click OK to close the dialog boxes.
8. Save the workspace file.
9. Select Tools»Deploy TestStand System in the sequence editor to launch the TestStand Deployment Utility.
10. On the System Source tab, enable the Deploy Files»From TestStand Workspace File option.
11. On the Installer Options tab, use the Installer Directory field to specify the directory to which the deployment utility copies the installer for the system.
12. Click the File Browse button located next to the From TestStand Workspace File control.
14. Click Open.
15. Click the Distributed Files tab.
16. Click Yes in the dialog box that launches to allow the deployment utility to analyze the workspace file and dependent files.
17. 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.

18. In the File Properties section on the Distributed Files tab, enable the **Create Program Item** option.

19. Enter `Simple CVI UI` in the associated string field to add a shortcut menu item for `TestExec.exe`.

20. On the **Installer Options** tab, enable the **Install TestStand Engine** option.

21. Click **Save** to save the build as `SimpleCVIUI.tsd`.

22. Click the **Build** button to create the installer.

23. To use the installer, copy all the files from the installer directory you specified on the Installer Options tab to a DVD or CD or to a shared directory on a network.

24. On the target computer, insert the DVD or CD or connect to the network and run the `setup.exe` application to start the installer.

25. When the installation completes, select **Start»All Programs»<my TestStand system>»Simple CVI UI** to verify the installation.
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 TestStand Sequence Editor or a TestStand 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 Microsoft 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 and the `<TestStand Public>\Components\Translators` directory. When you run the TestStand
sequence file translator example, the example copies the example translator DLL to the `<TestStand Public>\Examples\SequenceFileTranslators` directory, and you must restart the application before you can use the translator. To create new translator DLLs, add the project for the translator in the `<TestStand Public>\Components\Translators` directory and 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 whether 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.
Chapter 15  Sequence File Translators

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 whether 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. When 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 File and select `SampleTestFile.xml`, `SampleTestFile.lvtf` for the text version of the file for LabVIEW, `SampleTestFile.cvitf` for the text version of the file for LabWindows/CVI, or `SampleTestFile.vctf` 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.
When 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 whether the translator can load the file. In addition, TestStand displays the return value from the GetFileFormatVersion callback in reports the Workspace Documentation tool creates.

When 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.

When 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. When 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

When you place the custom sequence file translator DLL and support files in the `<TestStand Public>\Components\Translators` directory on the computer you use to build the deployment, the TestStand 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 when 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 to the *NI TestStand Help* for more information about the TestStand Deployment Utility.
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 unit under test (UUT) results to a database according to configuration settings, as shown in Figure A-1.
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 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 process models and their respective sequence files.

<table>
<thead>
<tr>
<th>Process Model</th>
<th>Process Model Sequence File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Model</td>
<td><code>&lt;TestStand&gt;\Components\Models\TestStandModels\SequentialModel.seq</code></td>
</tr>
<tr>
<td>Parallel Model</td>
<td><code>&lt;TestStand&gt;\Components\Models\TestStandModels\ParallelModel.seq</code></td>
</tr>
<tr>
<td>Batch Model</td>
<td><code>&lt;TestStand&gt;\Components\Models\TestStandModels\BatchModel.seq</code></td>
</tr>
</tbody>
</table>

You can create your own process models, or you can modify a copy of the default process models.

To modify the installed process models or to create a new process model, copy all 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. 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 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. Each entry point is a sequence in the process model file. The Configure and Execute menus of an application typically include the model entry points.

**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 direct a step that applies to the batch as a whole to run only once per batch instead of once for each UUT. You can also specify whether certain steps or groups of steps cannot run on more than one UUT at a time or whether 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.

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
Configure Report Options, Configure Database Options, or Configure Model Options. Configuration entry points do not affect UUTs tested in the execution.

- **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 the NI TestStand Help 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. Refer to the <TestStand Application Data> Directory section of Chapter 8, Customizing and Configuring TestStand, for more information about the configuration files in the <TestStand Application Data>\Cfg directory.

### 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. When 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. When 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 report for one UUT. Execution entry points do not call the TestReport callback
when you enabled the On-The-Fly Reporting option in the Report Options dialog box. The Sequential process model defines a report for a single UUT as a header, an entry for each step result, and a footer. When you do not want to override the entire TestReport callback, you can override the ModifyReportHeader, ModifyReportEntry, and ModifyReportFooter callbacks instead to customize the report.

Based on the settings in the Report Options dialog box, the TestReport callback determines whether 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.

When you select the Sequence option in the Report Options dialog box, the TestReport callback calls the AddReportBody sequence in reportgen_html.seq, reportgen_txt.seq, or reportgen_xml.seq to build the report body. If the report format is HTML or text, the report generator uses a series of sequences with steps that recursively process the result list for the execution. When 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 the ATMLSupport.dll in the LabWindows/CVI project. The project and the source code for the DLL built in LabWindows/CVI are located in the <TestStand>\Components\Models\TestStandModels\ATML directory.

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 when 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 when 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 when you enable the On-The-Fly Reporting option in the Report Options dialog box or when 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**—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 TestStand 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:

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 callback when you enable on-the-fly report generation or database logging.
8. Increments the UUT index.
9. Calls the PreUUT callback.
10. When no more UUTs exist, skips to step 20.
11. Determines the report file pathname, configures display settings, resets the report, and specifies the report location. When you use the \texttt{<UUTstatus>} macro in an expression to specify the report file path, TestStand uses a temporary file path as the report file pathname.
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, when enabled, for the new UUT.
14. Calls the MainSequence callback.
15. Calls the PostUUT callback.
17. Writes the UUT report to disk by appending to an existing file or creating a new file. Also adjusts the root tags when the report format is XML. When you use the \texttt{<UUTstatus>} macro in an expression to specify the report file path, TestStand adjusts the report file path as the UUT status information becomes available.
18. Calls the LogToDatabase callback.
19. Loops back to step 8.
20. Calls the PostUUTLoop callback.

**Single Pass**

Open \texttt{SequentialModel.seq} in the sequence editor and select the \textbf{Single Pass} sequence on the Sequences pane to examine the Sequential process model Single Pass Execution entry point, which performs the following significant actions:

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 callback when 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. When you use the `<UUTStatus>` macro in an expression to specify the report file path, TestStand uses a temporary file path as the report file pathname.

8. Starts on-the-fly report generation and database logging, when enabled, for the new UUT.

9. Calls the MainSequence callback.


11. Writes the UUT report to disk by appending to an existing file or creating a new file. Also adjusts the root tags when the report format is XML. When you use the `<UUTStatus>` macro in an expression to specify the report file path, TestStand adjusts the report file path as the UUT status information becomes available.

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 whether 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 in that 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. When 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. When 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. When the operator enters a serial number, the code for the dialog box stores the serial number in the
TestSocket.UUT.SerialNumber parameter. When 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:

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.
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 callback when 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. When no more UUTs exist, skips to step 15.

7. Determines the report file pathname, configures display settings, resets the report, and specifies the report location. When you use the <UUTStatus> macro in an expression to specify the report file path, TestStand uses a temporary file path as the report file pathname.

8. Starts on-the-fly report generation and database logging, when enabled, for the new UUT.

9. Calls the MainSequence callback.

10. Calls the PostUUT 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 when the report format is XML. When you use the <UUTStatus> macro in an expression to specify the report file path, TestStand adjusts the report file path as the UUT status information becomes available.

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:


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.


### 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 callback when you enable on-the-fly report generation or database logging.

2. Determines the report file pathname, configures display settings, resets the report, and specifies the report location. When you use the <UUTStatus> macro in an expression to specify the report file path, TestStand uses a temporary file path as the report file pathname.

3. Starts on-the-fly report generation and database logging, when 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 when the report format is XML. When you use the <UUTStatus> macro in an expression to specify the report file path, TestStand adjusts the report file path as the UUT status information becomes available.
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 whether 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 when all running test sockets are waiting for the WaitingForRequest parameter or when 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. When 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 the 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 in that 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.
When 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. When 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. When 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. When 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. When 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:

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. When 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. When 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 when 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 when 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 Status dialog box.

32. Waits for the Status dialog box. When 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 callback when 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. When 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. Determines the report file pathname, configures display settings, resets the report, and specifies the report location. When you use the \texttt{<UUTStatus>} macro in an expression to specify the report file path, TestStand uses a temporary file path as the report file pathname.

11. Starts on-the-fly report generation and database logging, when 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.


17. Calls the LogToDatabase callback.

18. Calls the SendControllerRequest utility sequence to synchronize with the controlling execution 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 when the report format is XML. When you use the \texttt{<UUTStatus>} macro in an expression to specify the report file path, TestStand adjusts the report file path as the UUT status information becomes available.

20. Calls the SendControllerRequest utility sequence to synchronize with the controlling executions before completing execution. The sequence waits until the controlling execution allows the test socket to complete execution.


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:

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 when 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 when 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.


**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` callback when 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. Determines the report file pathname, configures display settings, resets the report, and specifies the report location. When you use the `<UUTStatus>` macro in an expression to specify the report file path, TestStand uses a temporary file path as the report file pathname.

4. Starts on-the-fly report generation and database logging, when 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.


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 when the report format is XML. When you use the `<UUTStatus>` macro in an expression to specify the report file path, TestStand adjusts the report file path as the UUT status information becomes available.

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>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATMLSupport.dll</td>
<td>DLL that contains C functions the process model sequences call to generate ATML reports.</td>
</tr>
<tr>
<td>ATMLSupport.lib</td>
<td>Import library for ATMLSupport.dll.</td>
</tr>
<tr>
<td>banners.c</td>
<td>C source for functions that display status banners.</td>
</tr>
</tbody>
</table>
## Table A-2. Installed Support Files for the Process Model Files (Continued)

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BatchModel.seq</td>
<td>Entry point and Model callback sequences for the Batch process model.</td>
</tr>
<tr>
<td>batchuutdlg.c</td>
<td>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.</td>
</tr>
<tr>
<td>c_report.c</td>
<td>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.</td>
</tr>
<tr>
<td>ColorselectPopup.c</td>
<td>C source for the functions that display a dialog box in which you can select a color.</td>
</tr>
<tr>
<td>ColorSelectPopup.h</td>
<td>C header file that contains declarations for the function in ColorselectPopup.c.</td>
</tr>
<tr>
<td>main.c</td>
<td>C source for utility functions.</td>
</tr>
<tr>
<td>ModelOptions.c</td>
<td>C source for the functions that launch the Model Options dialog box and read and write the model options from and to disk.</td>
</tr>
<tr>
<td>modelpanels.h</td>
<td>C header file that contains declarations for the panels in modelpanels.uir.</td>
</tr>
<tr>
<td>modelpanels.uir</td>
<td>LabWindows/CVI user interface resource file that contains panels the functions in modelsupport2.dll use.</td>
</tr>
<tr>
<td>modelsupport2.dll</td>
<td>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. You must use the LabWindows/CVI Full Development System with the Interface to Win32 API feature installed to rebuild this DLL.</td>
</tr>
<tr>
<td>modelsupport2.fp</td>
<td>LabWindows/CVI function panels for the functions in modelsupport2.dll.</td>
</tr>
<tr>
<td>modelsupport2.h</td>
<td>C header file that contains declarations for the functions in modelsupport2.dll.</td>
</tr>
<tr>
<td>modelsupport2.lib</td>
<td>Import library in Visual C/C++ format for modelsupport2.dll.</td>
</tr>
</tbody>
</table>
## Table A-2. Installed Support Files for the Process Model Files (Continued)

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>modelsupport2.prj</td>
<td>LabWindows/CVI project that builds modelsupport2.dll and modelsupport2.cws.</td>
</tr>
<tr>
<td>ModelSupport.seq</td>
<td>Subsequences all process models use for report generation.</td>
</tr>
<tr>
<td>ParallelModel.seq</td>
<td>Entry point and Model callback sequences for the Parallel process model.</td>
</tr>
<tr>
<td>paralleluutdlg.c</td>
<td>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.</td>
</tr>
<tr>
<td>PropertyObject.xsd</td>
<td>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. Refer to the NI TestStand Help for more information about the PropertyObject.xsd file.</td>
</tr>
<tr>
<td>report.c</td>
<td>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.</td>
</tr>
<tr>
<td>report.h</td>
<td>C header file that contains declarations for the functions in report.c.</td>
</tr>
<tr>
<td>Report.xsd</td>
<td>XML schema that defines the content of TestStand XML reports. Refer to the NI TestStand Help for more information about the Report.xsd file.</td>
</tr>
<tr>
<td>reportgen_atml.seq</td>
<td>Subsequences that add the header, result entries, and footer for a UUT to an ATML report.</td>
</tr>
<tr>
<td>reportgen_html.seq</td>
<td>Subsequences that add the header, result entries, and footer for a UUT to an HTML report.</td>
</tr>
<tr>
<td>reportgen_txt.seq</td>
<td>Subsequences that add the header, result entries, and footer for a UUT to an ASCII text report.</td>
</tr>
<tr>
<td>reportgen_xml.seq</td>
<td>Subsequences that add the header, result entries, and footer for a UUT to an XML report.</td>
</tr>
</tbody>
</table>
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

When you want to customize report generation for a test station, modify the default 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

<table>
<thead>
<tr>
<th>Report Format</th>
<th>Header</th>
<th>Footer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATML</td>
<td>GetATMLReport sequence in <code>reportgen_atml.seq</code> generates the header when it generates the report body.</td>
<td>GetATMLReport sequence in <code>reportgen_atml.seq</code> generates the footer when it generates the report body.</td>
</tr>
<tr>
<td>HTML</td>
<td>AddReportHeader sequence in <code>reportgen_html.seq</code></td>
<td>AddReportFooter sequence in <code>reportgen_html.seq</code></td>
</tr>
<tr>
<td>Text</td>
<td>AddReportHeader sequence in <code>reportgen_txt.seq</code></td>
<td>AddReportFooter sequence in <code>reportgen_txt.seq</code></td>
</tr>
<tr>
<td>XML</td>
<td>AddReportHeader sequence in <code>reportgen_xml.seq</code></td>
<td>AddReportFooter sequence in <code>reportgen_xml.seq</code></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Report Format Generator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATML</td>
<td>GetATMLReport sequence in <code>reportgen_atml.seq</code> calls the <code>Get_Atml_Report</code> function in <code>ATML_Report.c</code> in the <code>ATMLSupport.prj</code> LabWindows/CVI project, located in the <code>&lt;TestStand&gt;\Components\Models\TestStandModels\ATML</code> directory</td>
</tr>
<tr>
<td>HTML sequence</td>
<td>AddReportBody sequence in <code>reportgen_html.seq</code>, which indirectly calls the <code>PutOneResultInReport</code> sequence for each result</td>
</tr>
<tr>
<td>HTML DLL</td>
<td><code>GetReportBody_CImplementation</code> function in <code>c_report.c</code> in the <code>modelsupport2.prj</code> LabWindows/CVI project</td>
</tr>
<tr>
<td>Text sequence</td>
<td>AddReportBody sequence in <code>reportgen_txt.seq</code>, which indirectly calls the <code>PutOneResultInReport</code> sequence for each result</td>
</tr>
</tbody>
</table>
### Table A-5. Report Generation Model Callbacks

<table>
<thead>
<tr>
<th>Section of Report to Alter</th>
<th>Model Callback Sequence to Override</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>ModifyReportHeader</td>
</tr>
<tr>
<td>Footer</td>
<td>ModifyReportFooter</td>
</tr>
<tr>
<td>Each step result</td>
<td>ModifyReportEntry</td>
</tr>
<tr>
<td>TestStand does not call this callback when 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.</td>
<td></td>
</tr>
<tr>
<td>Entire report</td>
<td>TestReport</td>
</tr>
<tr>
<td>Batch header</td>
<td>ModifyBatchReportHeader</td>
</tr>
<tr>
<td>Batch footer</td>
<td>ModifyBatchReportFooter</td>
</tr>
<tr>
<td>Each test socket result</td>
<td>ModifyBatchReportEntry</td>
</tr>
<tr>
<td>Entire batch report</td>
<td>BatchReport</td>
</tr>
</tbody>
</table>

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, when 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, when 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, when 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 units under test (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 to create a unique flow of execution for each UUT.

The 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. When 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 many of the Synchronization step types to specify a name, lifetime settings, and timeout settings for 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 Synchronization objects share the same name space, you cannot create Synchronization objects with the same name. Synchronization object names are not case sensitive.

When 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.

When 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 Microsoft Windows system security to allow users to launch and access the `TSAutoMgr.exe` server remotely. Refer to the Setting Up TestStand for
Accessing Synchronization Objects Remotely section in this appendix for information about how to use synchronization objects remotely.

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 when 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. When 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.

When you want to explicitly control the lifetime of the object reference or when 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 when 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 whether a timeout occurred.
Setting Up TestStand for Accessing Synchronization Objects Remotely

To create and access a synchronization object on another computer, you must configure Windows system security on both computers. In this appendix, the term remote computer refers to the computer that creates the synchronization object, and the term local computer refers to the computer that accesses the synchronization object.

Setting Windows System Security on the Remote Computer

You must configure the Distributed Component Object Model (DCOM) settings and Windows firewall settings on the computer that creates the synchronization object to allow users to access and launch the TestStand TSAutoMgr.exe server remotely.

DCOM Settings for the Remote Computer

Use the following information to configure the DCOM security permissions for the TSAutoMgr.exe server on the computer that creates the synchronization object.

Complete the following steps to configure the DCOM security permissions.

1. Log in as a user with administrator privileges.
2. 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.
3. On 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 and complete the following steps to set permissions.
   a. Click the **Edit Limits** button in the Access Permissions section to launch the Access Permission dialog box.
   b. Click **Add** to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the **ANONYMOUS LOGON** user.
   c. *(Windows 7/Vista)* Click **OK** to close the Select Users or Groups dialog box.
      *(Windows XP)* Click **OK** to close the Select Users, Computer, Groups dialog box.
   d. Select the user you added and enable **Remote Access** in the Permissions section.
   e. Click **OK** to close the Access Permission dialog box.
   f. *(Windows 7/Vista)* Click the **Edit Limits** button in the Launch and Activation Permissions section to launch the Launch and Activation Permission dialog box.
      *(Windows XP)* Click the **Edit Limits** button in the Launch and Activation Permissions section to launch the Launch Permission dialog box.
   g. Click **Add** to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the **ANONYMOUS LOGON** user.
   h. *(Windows 7/Vista)* Click **OK** to close the Select Users or Groups dialog box.
      *(Windows XP)* Click **OK** to close the Select Users, Computer, Groups dialog box.
   i. Select the user you added and enable **Remote Launch** and **Remote Activation** in the Permissions section.
   j. *(Windows 7/Vista)* Click **OK** to close the Launch and Activation Permission dialog box.
      *(Windows XP)* Click **OK** to close the Launch Permission dialog box.

7. Click **OK** to close the My Computer Properties dialog box.

8. On 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 AutoMgr** and select **Properties** from the context menu to launch the NI TestStand AutoMgr Properties dialog box.

10. On the **General** tab of the NI TestStand AutoMgr Properties dialog box, change the Authentication Level to **None**.

11. Click the **Security** tab of the NI TestStand AutoMgr Properties dialog box and complete the following steps to set permissions.
   a. **(Windows 7/Vista)** Select the **Customize** option and click the **Edit** button in the Launch and Activation Permission section to launch the Launch and Activation Permission dialog box.
      **(Windows XP)** Select the **Customize** option and click the **Edit** button in the Launch and Activation Permissions section to launch the Launch Permission dialog box.
   b. Click **Add** to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the **ANONYMOUS LOGON** user.
   c. **(Windows 7/Vista)** Click **OK** to close the Select Users or Groups dialog box.
      **(Windows XP)** Click **OK** to close the Select Users, Computer, Groups dialog box.
   d. Select the user you added and enable **Remote Launch** and **Remote Activation** in the Permissions section.
   e. **(Windows 7/Vista)** Click **OK** to close the Launch and Activation Permission dialog box.
      **(Windows XP)** Click **OK** to close the Launch Permission dialog box.
   f. Select the **Customize** option and click the **Edit** button in the Access Permissions section to launch the Access Permission dialog box.
   g. Click **Add** to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the **ANONYMOUS LOGON** user.
   h. **(Windows 7/Vista)** Click **OK** to close the Select Users or Groups dialog box.
      **(Windows XP)** Click **OK** to close the Select Users, Computer, Groups dialog box.
   i. Select the user you added and enable **Remote Access** in the Permissions section.
   j. Click **OK** to close the Access Permission dialog box.
12. On the **Identity** tab of the NI TestStand AutoMgr Properties dialog box, select the **This user** option and enter a user name and password or select the **The interactive user** option. Click **OK** to close the dialog box.

- If you select the The interactive user option, a user must be logged in at the physical computer console during remote sequence execution. You can automatically log on a user each time a computer reboots by setting registry keys. Refer to Microsoft Knowledge Base article 315231 at support.microsoft.com/kb/315231 for more information about activating automatic logon.

**Note** Regardless of whether you select the This user option or The interactive user option, the user account used on the remote computer must be an administrator account.

### Windows Firewall Settings for the Remote Computer

Use the following information to configure the Windows firewall on the computer that creates the synchronization object to allow TestStand to access the `TSAutoMgr.exe` application.

### Windows 7 Firewall Settings

Complete the following steps to configure the Windows firewall on the computer that creates the synchronization object.

1. Log in as a user with administrator privileges.
2. Navigate to **Windows Firewall** on the Windows Control Panel and click **Allow a program or feature through Windows Firewall** on the left panel of the Windows Firewall window to launch the Allowed Programs window.
3. Complete the following steps to add exceptions for the `REngine.exe` application with the firewall enabled.
   a. Click the **Allow another program** button to launch the Add a Program dialog box.
   b. Click **Browse** and select the client application, such as `<TestStand>Bin\SeqEdit.exe`.
   c. Click **Add** to close the Add a Program dialog box.
   d. Click **OK** to close the Allowed Programs window.
4. Click **Advanced settings** on the left panel of the Windows Firewall window to launch the Windows Firewall with Advanced Security window.
5. Complete the following steps to specify a security exception for DCOM.
   a. Click **Inbound Rules** on the left panel of the Windows Firewall with Advanced Security window to display the list of inbound rules.
   b. Click **New Rule** on the Actions pane to launch the New Inbound Rule Wizard.
   c. Select **Port** and click the **Next** button.
   d. Select **TCP**, select **Specific local ports**, enter **135** in the Specific local ports control, and click **Next**.
   e. Select **Allow the connection** and click **Next**.
   f. Ensure that the Domain, Private, and Public options have checkmarks and click **Next**.
   g. In the Name control, enter **DCOM** and click **Finish** to close the New Inbound Rule Wizard.

**Windows Vista Firewall Settings**

Complete the following steps to configure the Windows firewall on the computer that creates the synchronization object.

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 **TSAutoMgr.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.
   c. Click **Browse** and select `<TestStand>\Bin\TSAutoMgr.exe`.
   d. Click **OK** to close the Add a Program dialog box.
   e. Click the **Add port** button to launch the Add a Port dialog box.
   f. In the **Name** control, type **DCOM**.
   g. In the **Port Number** control, type **135**.
   h. Select the **TCP** option and click **OK** to close the Add a Port dialog box.

4. Click **OK** to close the Windows Firewall Settings dialog box.
Appendix B  Synchronization Step Types

Windows XP Service Pack 2 Firewall Settings
Complete the following steps to configure the Windows firewall on the computer that creates the synchronization object.

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 TSAutoMgr.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.
   c. Click Browse and select <TestStand>\Bin\TSAutoMgr.exe.
   d. Click OK to close the Add a Program dialog box.
   e. Click the Add Port button to launch the Add a Port dialog box.
   f. In the Name control, type DCOM.
   g. In the Port Number control, type 135.
   h. Select the TCP option and click OK to close the Add a Port dialog box.
4. Click OK to close the Windows Firewall dialog box.

Setting Windows System Security on the Local Computer
You must configure DCOM settings and Windows firewall settings on the computer that accesses the synchronization object remotely.

DCOM Settings for Local Computer
Complete the following steps to configure DCOM access permissions on the computer that accesses the synchronization object. You must restart TestStand for these changes to take effect.

1. Log in as a user with administrator privileges.
2. 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.
3. On the left pane of the Component Services window, select Component Services>Computers>My Computer.
Appendix B  Synchronization Step Types

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 and complete the following steps to set permissions.
   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. If ANONYMOUS LOGON does not appear in the user name list, click the Add button to add it.
   c. Click OK to close the Access Permissions dialog box.

You must also disable DCOM authentication for the client application if it is not the TestStand Sequence Editor (SeqEdit.exe) or one of the TestStand User Interfaces named testexec.exe. For client applications built with LabVIEW, disable DCOM authentication for the application by adding the following line to the INI file associated with the application, as in yourclient.ini where yourclient.exe is the name of your application:

```
ole.AuthnLevel=1
```

For all other client applications, disable DCOM authentication for the application by adding the following registry entry, where yourclient.exe is the name of your application:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AppID\yourclient.exe]
"AppID" ="{C31FD07F-DEAC-4962-9BBF-092F0F1BFF3C}"
```

Refer to Chapter 7, Effectively Using LabVIEW with TestStand, of the Using LabVIEW and LabWindows/CVI with TestStand manual for more information about how to best use LabVIEW features in a TestStand system.

Windows Firewall Settings for Local Computer

Use the following information to configure the Windows firewall on the local computer to allow the TSAutoMgr.exe application to access the TestStand client.

Windows 7 Firewall Settings

Complete the following steps to configure the Windows firewall on the local computer.

1. Log in as a user with administrator privileges.
2. Navigate to **Windows Firewall** on the Windows Control Panel and click **Allow a program or feature through Windows Firewall** on the left panel of the Windows Firewall window to launch the Allowed Programs window.

3. Complete the following steps to add exceptions for the REngine.exe application with the firewall enabled.
   a. Click the **Allow another program** button to launch the Add a Program dialog box.
   b. Click **Browse** and select the client application, such as `<TestStand>\Bin\SeqEdit.exe`.
   c. Click **Add** to close the Add a Program dialog box.
   d. Click **OK** to close the Allowed Programs window.

4. Click **Advanced settings** on the left panel of the Windows Firewall window to launch the Windows Firewall with Advanced Security window.

5. Complete the following steps to specify a security exception for DCOM.
   a. Click **Inbound Rules** on the left panel of the Windows Firewall with Advanced Security window to display the list of inbound rules.
   b. Click **New Rule** on the Actions pane to launch the New Inbound Rule Wizard.
   c. Select **Port** and click the **Next** button.
   d. Select **TCP**, select **Specific local ports**, enter 135 in the Specific local ports control, and click **Next**.
   e. Select **Allow the connection** and click **Next**.
   f. Ensure that the Domain, Private, and Public options have checkmarks and click **Next**.
   g. In the Name control, enter **DCOM** and click **Finish** to close the New Inbound Rule Wizard.

**Windows Vista Firewall Settings**

Complete the following steps to configure the Windows firewall on the local computer.

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 client application with the firewall enabled.
   a. Click the **Exceptions** tab.
   b. Click the **Add program** button to launch the Add a Program dialog box.
   c. Click **Browse** and select the client application, such as `<TestStand>\Bin\SeqEdit.exe`
   d. Click **OK** to close the Add a Program dialog box.
   e. Click the **Add port** button to launch the Add a Port dialog box.
   f. In the **Name** control, type **DCOM**.
   g. In the **Port Number** control, type **135**.
   h. Select the **TCP** option and click **OK** to close the Add a Port dialog box.

   Click **OK** to close the Windows Firewall Settings dialog box.

### Windows XP Service Pack 2 Firewall Settings

Complete the following steps to configure the Windows firewall on the local computer.

1. Log in as a user with administrator privileges.
2. Navigate to **Windows Firewall** on the Windows Control Panel 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 client application with the firewall enabled.
   a. Click the **Exceptions** tab.
   b. Click the **Add Program** button to launch the Add a Program dialog box.
   c. Click **Browse** and select the client application, such as `<TestStand>\Bin\SeqEdit.exe`
   d. Click **OK** to close the Add a Program dialog box.
   e. Click the **Add Port** button to launch the Add a Port dialog box.
   f. In the **Name** control, type **DCOM**.
   g. In the **Port Number** control, type **135**.
   h. Select the **TCP** option and click **OK** to close the Add a Port dialog box.

4. Click **OK** to close the Windows Firewall dialog box.
Appendix B  Synchronization Step Types

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. The Resource Usage Profiler window does not provide a way to edit these 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, when 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 first-in first-out (FIFO) 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.

When all the threads that use a set of locks reside on the same computer and when all the locks in the set reside on that same computer, TestStand detects and reports a run-time error when 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` when 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**—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**—The Timeout Enabled setting for the Lock operation.

- **Step.TimeoutExpr**—The Timeout expression, in seconds, for the Lock operation.

- **Step.ErrorOnTimeout**—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**—The Number of Threads Waiting to Lock the Lock expression for the Get Status operation.

- **Step.Operation**—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**—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**—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**—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 when the Rendezvous operation times out.

• **Step.NameOrRefExpr**—The Rendezvous Name expression for the Create operation and the Rendezvous Name or Reference expression for all other Rendezvous operations.

• **Step.LifetimeRefExpr**—The object reference expression for the Rendezvous Reference Lifetime when you set the lifetime to Use Object Reference.

• **Step.TimeoutEnabled**—The Timeout Enabled setting for the Rendezvous operation.

• **Step.TimeoutExpr**—The Timeout expression, in seconds, for the Rendezvous operation.

• **Step.ErrorOnTimeout**—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**—The Number of Threads Per Rendezvous expression for the Create operation.
Appendix B  Synchronization Step Types

- **Step.NumThreadsWaitingExpr**—The Number of Threads Waiting for Rendezvous expression for the Get Status operation.

- **Step.Operation**—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**—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**—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. When 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 when 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**—The object reference expression for the Queue Reference Lifetime when you set the lifetime to Use Object Reference.

- **Step.TimeoutEnabled**—The Timeout Enabled setting for the Enqueue or Dequeue operation.

- **Step.TimeoutExpr**—The Timeout expression, in seconds, for the Enqueue or Dequeue operation.
Step.ErrorOnTimeout — 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 — The expression that specifies the maximum number of queue elements for the Create operation.

Step.MaxNumElementsOutExpr — The expression that specifies where to store the maximum number of queue elements for the Get Status operation.

Step.NumThreadsWaitingEnqueueExpr — The expression that specifies where to store the number of threads waiting to enqueue for the Get Status operation.

Step.NumThreadsWaitingDequeueExpr — The expression that specifies where to store the number of threads waiting to dequeue for the Get Status operation.

Step.Operation — 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 — 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 — 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 — The Boolean value that specifies to store a queue element by object reference instead of by value for the Enqueue operation.

Step.EnqueueLocation — 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 — 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**—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**—A Boolean value that specifies to remove the element from the queue when the step performs the Dequeue operation.

• **Step.WhichQueueExpr**—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 when 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**—The object reference expression for the Notification Reference Lifetime when you set the lifetime to Use Object Reference.

• **Step.TimeoutEnabled**—The Timeout Enabled setting for the Wait operation.

• **Step.TimeoutExpr**—The Timeout expression, in seconds, for the Wait operation.

• **Step.ErrorOnTimeout**—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.
Appendix B  Synchronization Step Types

- **Step.NumThreadsWaitingExpr**—The expression that specifies where to store the number of threads waiting on the notification for the Get Status operation.

- **Step.Operation**—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**—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**—The Boolean value that specifies to store the data by object reference instead of by value for the Set or Pulse operation.

- **Step.WhichNotificationExpr**—The expression that specifies where to store the array offset of the notification to which the Wait operation responds.

- **Step.IsSetExpr**—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**—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**—The AutoClear setting for the Set operation.

- **Step.PulseNotifyOpt**—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 the status and error properties of the Wait step. Therefore, when 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 when 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 when the Wait for Thread or Wait for Execution operation times out.

- **Step.TimeoutEnabled**—The Timeout Enabled setting for the Wait for Thread or the Wait for Execution operation.

- **Step.TimeoutExpr**—The Timeout expression, in seconds, for the Wait for Thread or the Wait for Execution operation.

- **Step.ErrorOnTimeout**—The Timeout Causes Run-Time Error setting for the Wait for Thread or the Wait for Execution operation.

- **Step.ThreadRefExpr**—The Thread Reference expression for the Wait for Thread operation when the Step.SpecifyBySeqCall property is False.

- **Step.SeqCallName**—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**—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**—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**—The time expression for the Time Interval or Time Multiple operation of the step.
Appendix B  Synchronization Step Types

- **Step.ExecutionRefExpr**—The expression that specifies a reference to the execution on which the Wait for Execution operation waits.

- **Step.SpecifyBySeqCall**—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** when the Wait step is waiting on a thread and to **False** when the Wait step is waiting on an execution.

- **AsyncId**—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. When 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 when the type of the outer section is One Thread Only. For example, when 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 when an Enter or Exit operation times out.
- **Step.TimeoutEnabled**—The Timeout Enabled setting for the Enter or Exit operation.
- **Step.TimeoutExpr**—The Timeout expression, in seconds, for the Enter or Exit operation.
- **Step.ErrorOnTimeout**—The Timeout Causes Run-Time Error setting for the Enter or Exit operation.
- **Step.Operation**—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**—The expression that specifies the section name for the Enter or Exit operation.
- **Step.SectionType**—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* when
  any Auto Scheduled Resource blocks within the Auto Schedule block
  time out.

- **Step.TimeoutEnabled**—The Timeout Enabled setting for the Auto
  Schedule operation.

- **Step.TimeoutExpression**—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 the 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**—A list of expressions that specify the
  lock alternatives the block can acquire before executing
  the steps within the block.
**Thread Priority**

Use the Thread Priority step, shown at left, to adjust the operating system priority of a TestStand thread so that the 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** — A value that specifies the operation for the step to perform. The valid values are \( 0 = \text{Set Thread Priority} \) and \( 1 = \text{Get Thread Priority} \).
- **Step.SetPriorityExpr** — The thread priority expression for the Set Thread Priority operation.
- **Step.GetPriorityExpr** — 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. When 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 when 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**—A Boolean value that specifies whether the Acquire operation automatically performs a release when the Acquire lifetime expires.

- **Step.LifetimeRefExpr**—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**—The Timeout Enabled setting for the Acquire operation.

- **Step.TimeoutExpr**—The Timeout expression, in seconds, for the Acquire operation.

- **Step.ErrorOnTimeout**—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**—The Initial Semaphore Count expression for the Create operation.

- **Step.NumThreadsWaitingExpr**—The Number of Threads Waiting to Acquire the Semaphore expression for the Get Status operation.

- **Step.Operation**—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**—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.
Appendix B  Synchronization Step Types

- **Step.InitialCountOutExpr**—The Initial Semaphore Count expression for the Get Status operation.
- **Step.AcquireLifetime**—A value that specifies the lifetime setting for the Acquire operation. The valid values are $0 = \text{Same as Sequence}$, $1 = \text{Same as Thread}$, and $2 = \text{Use Object Reference}$. The Acquire operation uses this setting only when Step.AutoRelease is True.
- **Step.CurrentCountExpr**—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 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**—A value that specifies the operation for the step to perform. The valid values are $0 = \text{Create}$, $1 = \text{Add Thread}$, $2 = \text{Remove Thread}$, and $3 = \text{Get Status}$.
- **Step.NameOrRefExpr**—Contains the Name expression for the Create operation and the Name or Reference expression for all other Batch operations.
- **Step.Lifetime**—A value that specifies the lifetime setting to use for the Create operation. The valid values are $0 = \text{Same as Sequence}$, $1 = \text{Same as Thread}$, $2 = \text{Use Object Reference}$, and $3 = \text{Same as Execution}$.
Appendix B  Synchronization Step Types

• **Step.LifetimeRefExpr**—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**—The Object Reference to Thread expression for the Add Thread and Remove Thread operations.

• **Step.OrderNumExpr**—The Order Number expression for the Add Thread operation.

• **Step.NumThreadsWaitingExpr**—The Number of Threads Waiting at Synchronized Sections expression for the Get Status operation.

• **Step.NumThreadsInBatchExpr**—The Number of Threads in Batch expression for the Get Status operation.

• **Step.DefaultBatchSyncExpr**—The Default Batch Synchronization expression for the Create operation.

• **Step.DefaultBatchSyncOutExpr**—The Default Batch Synchronization expression for the Get Status operation.

---

**CPU Affinity**

Symmetric multiprocessing (SMP) systems are computers that contain multiple CPUs, where a CPU can be a single core on a multi-core processor or a single-core processor on a multiprocessor computer. Windows and other modern operating systems can take advantage of SMP systems to achieve increased performance. SMP systems achieve better performance by executing multiple threads on multiple CPUs concurrently. Performance improvement can come through executing multiple processes concurrently as well as from executing multiple threads within a process concurrently. Generally, when an application is implemented with multiple threads, the operating system attempts to schedule each thread on a separate CPU when possible. Under some circumstances, this automatic scheduling can have a negative impact on application performance.

For instances in which the automatic scheduling of threads on CPUs negatively affects TestStand application performance, you can manually assign or constrain the TestStand process or specific threads to specific CPUs. Use the CPU Affinity step, shown at left, to adjust the CPU affinity of a process or thread to allow the process or thread to run on a given set of CPUs on the computer.
You can also use the Default CPU Affinity For Threads option on the Preferences tab of the Station Options dialog box to determine which CPUs the user interface thread uses and the default CPUs for new executions. Use the Sequence Call Advanced Settings window in the sequence editor or the Thread Settings dialog box or Execution Settings dialog box in user interfaces to set which CPUs to use when executing a sequence in a new thread or execution.

Refer to the NI TestStand Help for more information about using TestStand on SMP systems.

**Step Properties**

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

- **Step.Operation**—A value that specifies the operation for the step to perform. The valid values are 0 = Get System CPU Affinity, 1 = Get Process CPU Affinity, 2 = Set Process CPU Affinity, 3 = Get Thread CPU Affinity, and 4 = Set Thread CPU Affinity.

- **Step.Expr**—An expression value that specifies or stores the CPU affinity for the system, process, or thread. For a get operation, the expression specifies a variable in which to store the value. For a set operation, the expression specifies a CPU affinity value. The CPU affinity is a numeric value where each bit represents a CPU. The lowest-order bit represents the first CPU. For example, a value of 12 (0b1100) represents CPUs 3 and 4 on a quad-core system.
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. Use the Property Loader step type to import property and variable values from a file or database during an execution.

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.

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**—A string expression that contains the name of the data link to open.
- **Step.DatabaseHandle**—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**—The number of records in a page for the SQL statement.
- **Step.CommandTimeout**—The amount of time, in seconds, TestStand waits while attempting to issue a command to the open database connection.
- **Step.CacheSize**—The cache size for the SQL statement.
- **Step.MaxRecordsToSelect**—The maximum number of records the SQL statement can return.
- **Step.CursorType**—The cursor type the SQL statement uses.
- **Step.CursorLocation**—Specifies where the data source maintains cursors for a connection.
- **Step.MarshalOptions**—The marshal options for the updated records in the SQL statement.
- **Step.LockType**—The lock type for the records the SQL statement selects.
- **Step.CommandType**—The command type of the SQL statement.
- **Step.DatabaseHandle**—The name of the variable or property that contains the database handle with which you open the SQL statement.
- **Step.StatementHandle**—The numeric variable or property the step type assigned as the value of the SQL statement handle.
Appendix C  Database Step Types

- **Step.SQLStatement**—A string expression that contains the SQL command.
- **Step.NumberOfRecordsSelected**—The numeric variable or property to which the step assigns the number of records the SQL statement returns.
- **Step.RequiresParameters**—Specifies whether the SQL statement requires input or output parameters to execute. When this property is False, the step immediately executes the SQL statement. When this property is 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. When 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 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**—A string expression that contains the name of the SQL statement on which to operate.
- **Step.RecordToOperateOn**—The record on which to operate. Valid values are 0 = New, 1 = Current, 2 = Next, 3 = Previous, and 4 = Index.
- **Step.RecordIndex**—The index of the record on which to operate when you set the `Step.RecordToOperateOn` property to fetch a specific index.
- **Step.Operation**—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**—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**—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**—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**—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 whether TestStand writes NULL to the column instead of the value in the Data expression property.
  - **Status**—The error code TestStand returns for the Get or Set operation.
  - **Direction**—An enumerated value that specifies the parameter direction as In, Out, In/Out, or Return.
  - **Type**—An enumerated value that specifies the parameter value as String, Number, Boolean, or Date/Time.
  - **Size**—The maximum size of a string parameter.

• **Step.SQLStatement**—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 database management system (DBMS) at run time.

**Note**  The Property Loader step type supports only ANSI file formats.

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. When you use step properties to hold the limit values, include a Property Loader step in the Setup step group of the sequence to initialize
Appendix C  Database Step Types

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.

Refer to the Database Known Issues topic in the NI TestStand Help for information about known issues when using DBMS client software with TestStand.

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>  Variable Value
Count  100

<FileGlobals>  Variable Value
Count  99

<StationGlobals>  Variable Value
Power_On  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 to 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>
<thead>
<tr>
<th>STEPNAME</th>
<th>LIMITS_ HIGH</th>
<th>LIMITS_ LOW</th>
<th>LIMITS_ STRING</th>
<th>POWER_ON</th>
<th>COUNT</th>
<th>SEQUENCE NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voltage at Pin A</td>
<td>9.0</td>
<td>11.0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Phone Test.seq</td>
</tr>
<tr>
<td>Voltage at Pin B</td>
<td>8.5</td>
<td>9.5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Phone Test.seq</td>
</tr>
<tr>
<td>Self Test Output</td>
<td>—</td>
<td>—</td>
<td>&quot;SYS OK&quot;</td>
<td>—</td>
<td>—</td>
<td>Phone Test.seq</td>
</tr>
<tr>
<td>&lt;Locals&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>100</td>
<td>Phone Test.seq</td>
</tr>
<tr>
<td>&lt;File Globals&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>99</td>
<td>Phone Test.seq</td>
</tr>
<tr>
<td>&lt;Station Globals&gt;</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>False</td>
<td>—</td>
<td>Phone Test.seq</td>
</tr>
</tbody>
</table>
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 to 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**—The total number of values the step loaded from the file or database.
- **Step.Result.NumPropertiesApplied**—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**—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**—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**—The name or number of the column on which to perform the comparison.
Appendix C  Database Step Types

- **Data**—The expression TestStand evaluates at run time to compare against the column value.
- **FormatString**—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**—An enumerated value that specifies the parameter direction as In, Out, In/Out, or Return.
- **Type**—An enumerated value that specifies the parameter type as String, Number, Boolean, or Date/Time.
- **Size**—The maximum size of a string parameter.
- **WriteNull**—Not used.
- **Status**—Not used.

- **Step.PropertiesListSource**—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**—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**—The name of the property or variable to which TestStand assigns a value.
  - **PropertyType**—The scope of the property or variable. Valid values are 0 = Step, 1 = Local, 2 = File Global, and 3 = Station Global.
  - **DataType**—The TestStand type of the property. Valid values are 1 = String, 2 = Boolean, and 3 = Number.
  - **ColumnName**—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**—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**—The name of the variable or property that contains the SQL statement handle the step uses at run time to load values.

– **SQLStatement**—The SQL statement the Edit Property Loader dialog box uses to populate ring controls that contain column names.

– **StepNameColumn**—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 whether TestStand appends the data type name of the property to the column name when selecting a property from the available list.

– **FilterUsingColumnList**—Specifies whether the step loads only the rows that match the specific column value.

– **MaxColumnSize**—The maximum number of characters for a column name.

• **Step.File**—The file and settings for importing property values from a file to a sequence file. The File step property contains the following subproperties:
  – **Path**—A literal pathname for the data file.
  – **DecimalPoint**—The type of decimal point the file uses.
  – **FileExpr**—A pathname expression TestStand evaluates at run time.
  – **Format**—The type of delimiters in the file and the file type. Valid values are Tab, Comma, or Excel.
  – **Start.MarkerExpr**—The expression for the starting marker.
  – **End.MarkerExpr**—The expression for the ending marker.
  – **Skip**—The string that causes the step type to ignore the row when the string appears at the beginning of the row.
  – **MapColumnsUsingFirstRow**—Specifies whether 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**—The names of the properties into which the step loads the values when `Step.File.MapColumnsUsingFirstRow` is False.
- **Step.SequenceFile**—The path to the sequence file to import properties to.

- **Step.Sequence**—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 whether 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 National Instruments devices and other popular instruments. 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 Component Object Model (COM) technology. The IVI step types support IVI-C class-compliant instrument drivers and support IVI-COM class-compliant instrument drivers when you install the IVI-COM Adapter component of the IVI Component Package included in the NI Device Driver DVD. 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. When 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 when 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**—The measurement values for the Read and Fetch operations. The property data type is NI_IviSinglePoint or NI_IviWave.
- **Step.LogicalName**—The logical name expression.
- **Step.InstrOperation**—A value that specifies the operation you configured the step to perform.
- **Step.SettingsSource**—The name of the property or variable where the step loads and stores the settings for the operation.
- **Step.Configuration**—The settings for the Configure operation. The data type of this property is NI_IviDmmConfig.
- **Step.SoftFrontPanel**—The settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.
• **Step.Readings**—The settings for the Read and Fetch operations.

• **Step.GetInfo**—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**—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**—The logical name expression.
- **Step.InstrOperation**—A value that specifies the operation you configured the step to perform.
- **Step.SettingsSource**—The name of the property or variable where the step loads and stores the settings for the operation.
Appendix D   IVI Step Types

- **Step.Configuration**—The settings for the Configure operation. The data type of this property is NI_IviScopeConfig.
- **Step.SoftFrontPanel**—The settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.
- **Step.Readings**—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**—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 when 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**—The logical name expression.
- **Step.InstrOperation**—A value that specifies the operation you configured the step to perform.
- **Step.SettingsSource**—The name of the property or variable where the step loads and stores the settings for the operation.
- **Step.Configuration**—The settings for the Configure operation. The data type of this property is NI_IviFgenConfig.
- **Step.SoftFrontPanel**—The settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.
- **Step.GetInfo**—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**—The measurement values for the Measure operation. The property data type is an array of NI_IviSinglePoint.
- **Step.LogicalName**—The logical name expression.
- **Step.InstrOperation**—A value that specifies the operation you configured the step to perform.
- **Step.SettingsSource**—The name of the property or variable where the step loads and stores the settings for the operation.
- **Step.Configuration**—The settings for the Configure operation. The data type of this property is NI_IviDCPowerConfig.
- **Step.SoftFrontPanel**—The settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.
- **Step.Readings**—The settings for the Measure operation. The data type of this property is NI_IviDCPowerReadings.
- **Step.GetInfo**—The settings for the Get Information operation.
- **Step.ResetOutputProtection**—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](http://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.
Appendix D  IVI Step Types

- **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**—The logical name expression.
- **Step.InstrOperation**—A value that specifies the operation you configured the step to perform.
- **Step.SettingsSource**—The name of the property or variable where the step loads and stores the settings for the operation.
- **Step.IviOperation**—A value that specifies the operation you configured the step to perform for IVI Switching mode.
- **Step.ConnectDisconnect**—The settings for the Connect/Disconnect operation.
- **Step.SoftFrontPanel**—The settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.
- **Step.GetInfo**—The settings for the Get Information operation.
- **Step.ScanningConfig**—The settings for the Configure Scan operation.
- **Step.Wait**—The settings for the Wait operation.
- **Step.Configure**—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**—The logical name expression.
- **Step.InstrOperation**—A value that specifies the operation you configured the step to perform.
- **Step.SettingsSource**—The name of the property or variable where the step loads and stores the settings for the operation.
- **Step.SoftFrontPanel**—The settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.
- **Step.Init**—The settings for the Init operation.
- **Step.SelfTest**—The settings for the Self Test operation.
- **Step.SessionInfo**—The settings for the Get Session Info operation.
Appendix D  IVI Step Types

- **Step.RevisionQuery**—The settings for the Revision Query operation.
- **Step.ErrorQuery**—The settings for the Error Query operation.
- **Step.ErrorInfo**—The settings for the Get Error Info operation.
- **Step.Attributes**—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 programatically linking instrument driver VIs on the block diagram. LabVIEW Plug and Play instrument drivers usually use Virtual Instrument Software Architecture (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 LabVIEWIOReference 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 when 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. Refer to Chapter 7, Effectively Using LabVIEW with TestStand, of the Using LabVIEW and LabWindows/CVI with TestStand manual for more information about how to best use LabVIEW features in a TestStand system.

Check Remote System Status

Use the Check Remote System Status step type to determine whether LabVIEW is running on a remote computer and whether 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**—The computer name or IP address of the remote computer. The **Step.RemoteHostByExpr** property specifies whether the step interprets this property value as an expression or as a string.
- **Step.RemoteHostByExpr**—When this property is True, the step interprets the **Step.RemoteHost** property value as an expression. When this property is False, the step interprets the value as a string.
- **Step.PortNumber**—The remote host port number.
- **Step.Timeout**—The number of seconds to wait to connect to the remote computer.
- **Step.ServerCheckExpr**—Specifies where to store a Boolean value that indicates whether 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**—The computer name or IP address of the remote computer. The Step.RemoteHostByExpr property specifies whether the step interprets this property value as an expression or as a string.
- **Step.RemoteHostByExpr**—When this property is True, the step interprets the Step.RemoteHost property value as an expression. When this property is False, the step interprets the value as a string.
- **Step.PortNumber**—The remote host port number.
- **Step.Timeout**—The number of seconds to wait to connect to the remote computer.
- **Step.VIModule**—The settings for the VI the step calls.

Deploy Library

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

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

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

**Note**  If you execute VIs in the context of a LabVIEW project, you can also enable the Auto Deploy Shared Variables on First Load of any VI in a Project option and the Auto Undeploy Shared Variables on Last Unload of all VIs in a Project option in the LabVIEW Adapter Configuration dialog box to automatically deploy and undeploy shared variables defined in the LabVIEW project.
Refer to the Network-Published Shared Variables section of Chapter 7, Effectively Using LabVIEW with TestStand, of the Using LabVIEW and LabWindows/CVI with TestStand manual for more information about shared variables. Refer to the NI TestStand Help for more information about the LabVIEW Adapter Configuration dialog box.

### Step Properties

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

- **Step.Operation**—Specifies whether the step deploys or undeploys shared variables. Valid values are 0 = Deploy and 1 = Undeploy.
- **Step.Libraries**—An expression for the path of the LabVIEW packed project library file, LabVIEW library file, or LabVIEW project file on the local computer to deploy to or undeploy from the target computer. The packed 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 at ni.com/support includes the following resources:
  - **Self-Help Technical Resources**—For answers and solutions, visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Standard Service Program Membership**—This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support as well as exclusive access to on demand training modules via the Services Resource Center. NI offers complementary membership for a full year after purchase, after which you may renew to continue your benefits.

    For information about other technical support options in your area, visit ni.com/services, or contact your local office at ni.com/contact.

- **Training and Certification**—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.

- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.
If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.
Index

A
aborting executions, 3-7
Action step, 4-13
ActiveX controls
  interface pointer, obtaining, 9-17
  Microsoft Visual C++, using with, 9-16
  Multiple Document Interface (MDI)
    application, creating, 9-15
    using with DLLs, 5-6
ActiveX/COM Adapter, 1-7, 5-14
  registering and unregistering servers, 5-15
  using ActiveX/COM servers with TestStand, 5-15
ActiveX/COM server
  debugging, 5-14
  registering and unregistering, 5-15
  adapters. See module adapters
Additional Results
  panel, 4-5
  step, 4-26
Allow Automatic Type Conflict Resolution options. See types
analyzing sequence files. See TestStand Sequence Analyzer
Application Manager control, 9-3
  command-line arguments, 9-29
  event handling, 9-18
application manifests, 9-34
application settings
  configuration file location, 9-30
  custom, adding, 9-31
  persisting, 9-30
application styles for user interfaces
  multiple window, 9-27
  no visible window, 9-29
  single window, 9-26
array property, 1-8
arrays
  dynamic array sizing, 12-3
  empty, 12-4
  specifying array sizes, 12-2
assemblies in .NET, debugging, 5-11
ATML Test Results and Session Information schema, 6-18
Authenticode signatures, 9-32
Auto Schedule step, B-24
automatic result collection, 1-16

B
Batch process model, A-5
  Configuration entry point, A-24
  Engine callbacks, A-27
  hidden Execution entry point, A-23
  Model callbacks, A-24
  Single Pass – Test Socket entry point, A-32
  Single Pass Execution entry point, A-31
  Test UUTs – Test Socket entry point, A-29
  Test UUTs Execution entry point, A-27
  utility subsequences, A-27
Batch reports, 6-17
Batch Specification step, B-28
Batch Synchronization step, B-22
  mismatched sections, B-23
  nested sections, B-23
  One Thread Only section, B-23
  Parallel section, B-23
  Serial section, B-22
  step properties, B-24
  synchronized sections, B-22
Break step, 4-17
  built-in sequence properties, 1-14
  built-in step properties, 4-3
ComboBox control
- description (table), 9-5
- command connections, 9-9
  - invoking, 9-10
- command-line arguments, 9-29
- command-line arguments for user interfaces, 9-29
- configuration
  - See also customizing
  - Configure menu, 8-13
  - module adapters, 1-7, 5-1
  - remote sequence execution, 5-18
  - sequence editor or user interface startup options (table), 8-11
  - user interface application settings configuration file, location, 9-30
- Configuration entry point, 1-16, 10-4
  - Configure Database Options, 10-4
  - Configure Model Options, 10-4
  - Configure Report Options, 10-4
- process models
  - Batch process model, A-24
  - Parallel process model, A-15
  - Sequential process model, A-6
- configuration file
  - loading .NET CLR (example), 5-13
  - user interface application settings, location, 9-30
- Configure Database Options Configuration entry point, 10-4, A-6
- Configure menu, 8-13
- Configure Model Options Configuration entry point, 10-4, A-6
- Configure Report Options Configuration entry point, 10-4, 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, 4-17
- conventions used in the manual, iv
- CPU Affinity step, B-29
- custom application settings, adding, 9-31
- custom data types, 1-9, 1-11
  - creating and modifying, 12-7
- custom properties
  - See also step properties
  - built-in step types, 3-9
  - lifetime of, 3-3
  - result properties, 3-9
  - step properties, 1-11, 13-9
  - step type, 13-9
- 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, 9-32
- custom user privileges, 7-3
- customizing
  - reports using expressions, 6-15
  - string resource files, 8-9
  - TestStand components (table), 8-1
  - Tools menu, 8-2
Index

D
data links, 6-4, 6-11
  specifying (tutorial), 6-12
Data Operation step, C-4
data source, 4-10
data types
  categories, 12-2
  common properties, 12-8
  creating custom data types (tutorial), 12-7
  creating instances from context menus
    (tutorial), 12-1
  custom properties, 12-9
  displaying, 12-4
  modifying, 12-4
  named, 12-2
  simple, 12-2
  standard named, 1-10, 12-5
    CommonResults, 12-6
      version 3.5.0.1 and 3.5.0.100
        discrepancies (note), 12-6
      Error, 12-6
      Expression, 12-7
Path, 12-7
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, 4-25, 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
data base 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
debug
  DLLs, 5-8
  executions, 3-5
  HTBasic Adapter, 5-15
  .NET assemblies, 5-11
  panes, 3-6
Deploy Library step, E-2
deployment, 1-2, 2-6
  See also TestStand Deployment Utility
diagnostic tools (NI resources), F-1
directory structure, 8-2
  <TestStand Application Data> directory,
    8-7
    subdirectories (table), 8-7
  <TestStand Local Application Data>
    directory, 8-8
    subdirectories (table), 8-9
Index

<TestStand Public> directory, 8-5
  subdirectories (table), 8-5
<TestStand> directory, 8-3
  Components directory, 8-4
    subdirectories (table), 8-4
  subdirectories (table), 8-3
  read-only files, copying to modify, 8-6
  search paths, 5-2

DLLs
  ActiveX controls, using, 5-6
  debugging, 5-8
    DLL functions, options for stepping out of (table), 5-9
    LabVIEW 8 or later, 5-6
    LabVIEW shared libraries, 5-9
  Microsoft Foundation Class (MFC) Library, using, 5-6
  parameter information, reading, 5-7
  subordinate, loading, 5-6
  translator DLL, creating, 15-2
  using with TestStand, 5-6

Do While step, 4-17

drivers
  LabVIEW Plug and Play, D-11
  LabWindows/CVI Plug and Play, D-11
  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, 4-15

Else step, 4-15

empty arrays, 12-4

End step, 4-18

Engine callbacks, 1-16, 3-15, 10-4, A-10
  Batch process model, A-27
  caveats, 10-8
  execution, using during, 3-15
  Parallel process model, A-16
  predefined (note), 10-5
  Sequential process model, A-10
  table, 10-5

engine. See TestStand Engine

entry points
  Configuration, 1-16, 10-4, A-6
  entry point sequences, 10-3
  Execution, 1-16, 3-4, A-5

errors
  run-time, 1-12, 3-19
    caution, 4-4
    handling interactively, 3-20
  step failure, 3-19

event handling, 9-17

  creating event handlers in specific ADEs (table), 9-18
  DisplayExecution event, 9-19
  DisplaySequenceFile event, 9-19
  ExitApplication event, 9-18
  ReportError event, 9-19
  shutdown, 9-20
  startup, 9-20
  Wait event, 9-18

examples (NI resources), F-1

Execution
  object, 1-17
  window, 3-3

Execution entry point, 1-16, 3-4

hidden
  Batch process model, A-23
  Parallel process model, A-15
  Parallel process model, A-12
  Sequential process model, A-5
Index

Single Pass, 10-4, A-6
  Batch process model, A-31
  Parallel process model, A-18
  Sequential process model, A-11
Single Pass – Test Socket entry point
  Batch process model, A-24, A-32
  Parallel process model, A-15, A-19
Test UUTs, 10-4, A-5
  Batch process model, A-27
  Parallel process model, A-17
  Sequential process model, A-10
Test UUTs – Test Socket entry point
  Batch process model, A-23, A-29
  Parallel process model, A-15, A-17

execution pointer, 3-1
executions, 3-1
  aborting, 3-7
  debugging, 3-5
    Call Stack pane, 3-6
    Output pane, 3-6
    Threads pane, 3-6
    Variables pane, 3-6
    Watch View pane, 3-6
direct, 3-4
executing sequences, 3-4
Execution entry points, using, 3-4
Execution object, 1-17
execution pointer, 3-1
Execution window, 3-3
interactive, 3-5
order of execution (table), 3-15
remote sequence execution, 5-16
run-time copy (note), 3-1
run-time errors, 3-19
step execution, 3-15
step status, 3-18
suspended or stopped executions within code modules, 3-7
terminating, 3-7, 3-19
ExecutionView Manager control, 9-4
  connecting views, 9-7

ExpressionEdit control
  description (table), 9-6
  expressions, 1-10
    context-sensitive editing, 1-10
    customizing reports, 6-15
    Sequence Call step, 4-14
    sequence context, using, 3-2
Expressions panel, 4-4

F

file types
  project, 2-5
  project file, 2-5
  string resource files, creating, 8-9
  type palette, 11-8
  user interface application settings
    configuration file, location, 9-30
    workspace, 2-5
    workspace options file, 2-6
files
  deployment
    collecting for, 14-4
    filtering, 14-4
    organizing for, 14-3
  read-only, copying to modify, 8-6
filtering files for deployment, 14-4
Flow Control step types, 4-15
  Break, 4-17
  Case, 4-18
  Continue, 4-17
  Do While, 4-17
  Else, 4-15
  Else If, 4-15
  End, 4-18
  For, 4-16
  For Each, 4-16
  Goto, 4-18
  If, 4-15
  Select, 4-17
  While, 4-17
For Each step, 4-16
For step, 4-16
Front-End callbacks, 10-9
    modifying, 10-9
FTP Files step, 4-25

G
General panel, 4-3
generic types support for the .NET Adapter, 5-11
Goto step, 4-18

H
hidden Execution entry point
    Batch process model, A-23
    Parallel process model, A-15
HTBasic Adapter, 1-7, 5-15
    debugging subroutines, 5-15

I
If step, 4-15
image connections, 9-11
include directories in Visual Studio, 5-5
information source connections, 9-10
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
    logical name, D-1
    session, D-1
    soft front panel, D-2
Instrument Driver Network, D-1
instrument drivers, D-1
    NI resources, F-1
    Plug and Play, D-11
        LabVIEW, D-11
        LabWindows/CVI, D-11
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-7
    IVI Tools, D-10

K
KnowledgeBase, F-1

L
Label control
description (table), 9-6
Label step, 4-19
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-11
Index

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-2
LabVIEW Adapter, 1-7, 5-3
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-11
  TestStand User Interface (UI) Controls, using, 9-14
  using TSUtil library (table), 9-22
LabWindows/CVI Adapter, 5-4
legacy code template, 13-8
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, 2-3
lifetime, 3-3
sequence local, 1-13
localizing user interface applications, 9-25
localization functions (table), 9-26
Lock step, B-14
Logging property in the sequence context, 6-7
logging results to a database, on-the-fly, 6-11
loop results, 3-14
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
menu open notification methods in ADEs (table), 9-24
menus
  Configure, 8-13
  Tools, 8-2
    modifying, 8-2
    updating, 9-24
merging sequence files, 2-2
Message Popup step, 4-19
Microsoft
  Access
    example data link and result table setup, 6-12
    specifying data link and schema, 6-12
  ActiveX Data Objects (ADO), 6-2
  Internet Explorer
    security settings affecting measurement data display in XML reports, 6-22
  Microsoft Foundation Class (MFC)
    Library
      using in DLLs, 5-6
  Object Linking and Embedding Database (OLE DB), 6-2
  Visual C++
    TestStand User Interface (UI) Controls, using, 9-16
Visual Studio
  assembly references, adding, 9-23
TestStand API, accessing, 5-5, 5-13
  TestStand include directories, 5-5
TestStand User Interface (UI) Controls, using, 9-15
Visual Studio .NET 2003
  .NET assemblies, debugging, 5-11
Model callbacks, 1-16, 10-2
  Batch process model, A-24
  Parallel process model, A-15
  report generation (table), A-38
  Sequential process model, A-6
modifying types, 11-1
module adapters, 1-7
  ActiveX/COM, 1-7, 5-14
  built-in step types, using with, 4-5
  C/C++ DLL, 1-7, 5-4
  configuring, 5-1
  HTBasic, 1-7, 5-15
  LabVIEW, 1-7, 5-3
  LabWindows/CVI, 1-7, 5-4
  .NET, 1-7, 5-10
  overview, 1-7
  Sequence, 1-7, 4-14, 5-16
  source code templates, 5-2
  supported code modules, 5-1
monitoring variables, 3-2
Multiple Document Interface (MDI) application, creating, 9-15
Multiple Numeric Limit Test step, 4-10
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, 5-10
    generics, 5-11
    .NET assemblies, debugging, 5-11
    .NET CLR, 5-12
      configuration file for loading .NET CLR (example), 5-13
    .NET Framework, 5-12
      configuration file for loading .NET CLR (example), 5-13
  NI Switch Executive, D-7
    route specification string, D-8
  normal sequence files, 2-1
  normal sequences, 10-2
  Notification step, B-19
  Numeric Limit Test step, 4-8
  numeric value connections, 9-12

O
  object
    Execution, 1-17
    SequenceContext, 1-9
    Synchronization, B-1
      common attributes, B-3
      resources, profiling usage of. See Resource Usage Profiler
  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
  Expressions, 4-4
  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, 3-6
  debugging, 3-6
  Insertion Palette, 4-1
Output, 3-6
Sequence File window, 2-5
  Sequences pane, 2-4
  Steps pane, 2-3, 2-4
  Variables pane, 2-2, 2-3, 2-4, 2-5
Step Settings. See Step Settings pane
Steps, 2-3, 2-5
Threads, 3-6
Types, 2-2, 11-8
Variables, 2-2, 2-3, 2-4, 3-6, 12-4
View Types For, 11-7
Watch View, 3-6
Workspace, 2-5
Parallel process model, A-4, A-12
  Configuration entry point, A-15
  Engine callbacks, A-16
  Execution entry point, A-12
    hidden, A-15
  Model callbacks, A-15
  Single Pass – Test Socket entry point, A-19
  Single Pass Execution entry point, A-18
  Test UUTs – Test Socket entry point, A-17
  Test UUTs Execution entry point, A-17
utility subsequences, 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, 4-6
password protecting sequence files, 2-2
Plug and Play instrument drivers
  LabVIEW, D-11
  LabWindows/CVI, D-11
Post Actions panel, 4-4
Preconditions panel, 4-5
previous versions of TestStand, avoiding
  unwanted type version propagation to, 11-5
privileges
  accessing privilege settings
    any user, 7-3
    current user, 7-2
  custom privileges, defining, 7-3
  verifying, 7-3
Index

process models, 1-14
  architecture, A-1
  Batch. See Batch process model
  common features, A-4
  entry points, 1-15
  modifying process model sequence files, 10-1
  Parallel. See Parallel process model
  process flow (figure), A-2
  process model location (table), A-3
  process model sequence files, 2-1
  selecting the default process model, A-5
  Sequential. See Sequential process model
  station model, 1-15
  support files, installation (table), A-33
programming examples (NI resources), F-1
project file, 2-5
propagation of unwanted type versions, avoiding. See types
properties
  built-in
    sequence properties, 1-14
    shared (note), 3-1
    step properties, 4-3
  common to all step types, 13-2
  custom
    built-in step types, 3-9
    lifetime of, 3-3
    result, 3-9
    step properties, 1-11
    step type, 13-9
  expressions, using in, 1-10
  monitoring, 3-2
  object reference properties, 12-5
  property-array property, 1-8
  Result.Status standard values (table), 3-18
  shared built-in properties (note), 3-1
  single-valued property, 1-8
Property Browser panel, 4-5
Property Loader step, 4-25, C-5

Q
  Queue step, B-17

R
  reading parameter information, 5-7
  remote access for synchronization objects. See synchronization objects
  remote sequence execution, 5-16
  security configuration
    DCOM settings for local client, 5-25
    DCOM settings for remote server, 5-19, 5-20
    firewall settings for local client, 5-26
    Windows 7, 5-26
    Windows Vista, 5-27
    Windows XP, 5-28
    firewall settings for remote server, 5-23
    Windows 7, 5-23
    Windows Vista, 5-24
    Windows XP, 5-24
    setting Windows system security on local client, 5-25
    setting Windows system security on remote server, 5-19
    setting up TestStand as server, 5-18
    security permissions, 5-19
  Rendezvous step, B-16
  reports, 6-15
    Batch, 6-17
    customizing using expressions, 6-15
    failure chain, 6-17
    generating, 3-14
      functions and sequences, A-36
      header and footer (table), A-37
      Model callbacks (table), A-38
      on-the-fly, 6-17
      report body (table), A-37
    implementing, 6-14
property flags, 6-17
result collection, 3-8
schema
  ATML Test Results and Session Information, 6-18
  XML, 6-18
XML, 6-18
  associating style sheets, 6-19
displaying measurement data, 6-21
distributing, 6-21
example transformation, 6-20
schema, 6-18
transforming with XSLT, 6-19
ReportView control
description (table), 9-6
Requirements panel, 4-5
resource files. See string resource files
Resource Usage Profiler, B-14
result collection
custom result properties, 3-9
disabling, 3-8
loop results, 3-14
report generation, 3-14
standard result properties, 3-11
result collection, automatic, 1-16
Results property, 12-6
root interactive execution, 3-5
route specification string, D-8
Run Options panel, 4-3
Run VI Asynchronously step, E-2
run-time errors, 1-12, 3-19
calculation, 4-4
handling, 3-20

S

schema
  See also databases
  ATML Test Results and Session Information, 6-18
  specifying (tutorial), 6-12
  XML reports, 6-18
search directories. See search paths
search paths, 5-2
deployment, configuring for, 5-3
Edit Search Directories dialog box, 5-2
Select step, 4-17
Semaphore step, B-26
Sequence Adapter, 1-7, 4-14, 5-16
sequence analyzer. See TestStand Sequence Analyzer
Sequence Call step, 2-5, 4-13
expressions, 4-14
sequence context, 3-2
Logging property, 6-7
sequence editor, 1-2
configuring startup options (table), 8-11
Execution window, 3-3
panels. See panels
panes. See panes
Sequence File window, 2-4
Sequence Hierarchy window, 2-5
Workspace pane, 2-5
sequence execution, 1-17
  See also execution
sequence file global variables, 2-2
sequence file translators. See translators
Sequence File window, 2-4
  figure, 2-4
Steps pane, 2-5
Variables pane, 2-5
sequence files, 1-1, 1-14, 2-1, 11-8
analyzing. See TestStand Sequence Analyzer callbacks, 2-1
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 variables, 1-14, 2-2
merging, 2-2
normal, 2-1
password protecting, 2-2
process model, 2-1
processing with TestStand Deployment Utility, 14-7
special editing capabilities for process model sequence files
callback sequences, 10-2
entry point sequences, 10-3
normal sequences, 10-2
Station callbacks, 2-1
type definitions, 2-2
types of sequence files, 2-1
types, using with, 11-8
versioning custom sequence files, 15-3
views, 2-4
Sequence Hierarchy window, 2-5
sequence local variables, 1-13
SequenceContext object, 1-9
SequenceFileView Manager control, 9-4
sequences, 1-1, 1-12, 2-3
built-in properties, 1-14
callback sequences, 1-16
callbacks, 1-17
entry point, 10-3
executing directly, 3-4
local variables, 1-13, 2-3
Model callbacks, 10-2
multithreading options, 1-17
parameters, 1-13, 2-3
run-time errors, 1-12, 3-19
step groups, 1-12, 2-3
Sequences pane, 2-4
SequenceView control
description (table), 9-6
Sequential process model, A-4, A-5
Configuration entry points, A-6
Engine callbacks, A-10
Execution entry point, A-5
Model callbacks, A-6
Single Pass Execution entry point, A-11
Test UUTs Execution entry point, A-10
utility subsequences, A-9
Session Manager, D-2
shut down, user interface applications, 9-20
Single Pass – Test Socket entry point Batch process model, A-32
Parallel process model, A-19
Single Pass Execution entry point, 10-4, A-6
Batch process model, A-31
Parallel process model, A-18
Sequential process model, A-11
software (NI resources), F-1
source code control (SCC), 2-6
source code templates, 1-11, 5-2
See also code templates SQL. See structured query language (SQL)
standard named data types, 1-10, 12-5
CommonResults, 12-6
version 3.5.0.1 and 3.5.0.100 discrepancies (note), 12-6
Error, 12-6
Expression, 12-7
Path, 12-7
using, 12-5
standard result properties, 3-11
startup, user interface applications, 9-20
Statement step, 4-18
station global variables, 1-9, 11-9
Index

Station Globals window, 11-9
station model, 1-15
StatusBar control
description (table), 9-7
step groups, 1-12, 2-3
step properties
  built-in, 4-3
  Case, 4-18
  custom
    lifetime of, 3-3
    modifying, 4-5
  Pass/Fail Test, 4-7
  Select, 4-17
  String Value Test, 4-12
Step Properties dialog box
  Code Templates tab, 13-6
  Disable Properties tab, 13-6
  General tab, 13-3
    Advanced button menu options, 13-3
  Menu tab, 13-4
  Substeps tab, 13-4
  Version tab, 13-9
step results
  See also result collection
custom result properties (table), 3-9
exceptions, 3-11
loop results, 3-14
standard properties (table), 3-11
subsequences (table), 3-12
Step Settings pane
  Properties tab, 4-3
    Additional Results panel, 4-5
    Expressions panel, 4-4
    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, 3-18
failures, 3-19
terminations, 3-19
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
  shared, 3-9
  Insert Step submenu, 4-1
  Insertion Palette, 4-1
  module adapters, using with, 4-5
  multiple code templates, specifying, 13-9
  source code templates, 1-11, 5-2
steps, 1-1, 1-11
  execution, order of actions (table), 3-15
  failures, 3-19
  interactive execution, 3-5
  properties. See step properties status (table), 3-18
  step types. See step types
  substeps. See substeps
terminations, 3-19
Steps pane, 2-3, 2-5
string resource files
  creating, 8-9
  customizing, 8-9
  escape codes (table), 8-10
  formatting, 8-10
String Value Test step, 4-11
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), 3-13
substeps
   custom, 13-5
   Edit, 13-5
   Post-Step, 13-5
   Pre-Step, 13-4
Switching panel, 4-4
symmetric multiprocessing (SMP)
systems, B-29
Synchronization object, B-1
   common attributes, B-3
   lifetime, B-4
   name, B-3
   timeout, B-4
resources, profiling usage of. See Resource Usage Profiler
synchronization objects
   accessing remotely, B-5
security configuration
   DCOM settings for local computer, B-10
   DCOM settings for remote computer, B-5
firewall settings for local computer, B-11
   Windows 7, B-11
   Windows Vista, B-12
   Windows XP, B-13
firewall settings for remote computer, B-8
   Windows 7, B-8
   Windows Vista, B-9
   Windows XP, B-10
setting Windows system security on local computer, B-10
setting Windows system security on remote computer, B-5
Synchronization panel, 4-4
Synchronization step types, B-1
   Auto Schedule, B-24
   Batch Specification, B-28
   Batch Synchronization, B-22
   mismatched section, B-23
   nested section, B-23
   One Thread Only section, B-23
   Parallel section, B-23
   Serial section, B-22
   step properties, B-24
   synchronized section, B-22
   CPU Affinity, B-29
   Lock, B-14
   Notification, B-19
   Queue, B-17
   Rendezvous, B-16
   Semaphore, B-26
   Thread Priority, B-26
   Use Auto Scheduled Resource, B-25
   Wait, B-20
system deployment, 2-6

T
tables
   database result, 6-8
   creating (tutorial), 6-13
   creating with TestStand Database Viewer, 6-9
   default TestStand table schema, 6-8
   default result, creating, 6-9
   query, SQL statement data, 6-2
template
   code. See code templates
   legacy code, 13-8
   source code, 5-2
Index

step, 13-1
  custom step types, differences with, 13-1
terminating executions, 3-7, 3-19
test executive, 1-1
test modules. See code modules
test reports
  See also reports
Test step types. See built-in step types
Test UUTs – Test Socket entry point
  Batch process model, A-29
  Parallel process model, A-17
Test UUTs Execution entry point, 10-4, A-5
  Batch process model, A-27
  Parallel process model, A-17
  Sequential process model, A-10
tests, distributing, 14-11
TestStand
  API, accessing in Visual Studio, 5-5, 5-13
  TestStand include directories, 5-5
  architecture overview, 1-1
  building blocks
    callback sequences, 1-16
    custom data types, 1-9
    expressions, 1-10
    process models, 1-14
    properties, 1-8
    result collection, automatic, 1-16
    sequence executions, 1-17
    sequence files, 1-14
    sequences, 1-12
    standard data types, 1-9
    steps, 1-11
    variables, 1-9
    variables and properties, 1-9
  software components
    module adapters, 1-7
    sequence editor, 1-2
  TestStand Deployment Utility, 1-8
  TestStand Engine, 1-6

TestStand User Interface (UI)
  Controls, 1-6
  user interfaces, 1-4
  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, 1-1
  system components, 14-1
TestStand Database Viewer, 6-9, 6-13
  result tables, creating (tutorial), 6-13
TestStand Deployment Utility, 1-8, 2-6, 14-1
  build process, 14-4
  components to deploy, identifying, 14-2
  configuring, 14-3
  creating a deployment, 14-2
  deployment scenarios, common
dynamically called files, adding to workspaces, 14-12
TestStand Engine, deploying, 14-10
  user interfaces, distributing, 14-14
  workspaces, distributing tests from, 14-11
files
  collecting, 14-4
  filtering, 14-4
  organizing, 14-3
guidelines, 14-8
installer, creating, 14-3
NI components, installing, 14-7
path references, 14-7
search paths, configuring for deployment, 5-3
sequence files
  deploying custom, 15-4
  processing, 14-7
Index

system components, 14-1
VIs, processing, 14-6
TestStand Engine, 1-6
engine callbacks, 1-16
TestStand process models. See Batch process
model; Parallel process model; process
models; Sequential process model
TestStand Sequence Analyzer, 2-2
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, 9-26
command connections, 9-9
command-line arguments, 9-29
configuration file location, 9-30
connecting, 9-7
connections, specifying and
changing, 9-12
custom
  application settings, adding, 9-31
  user interfaces, documenting, 9-32
Editor applications, creating, 9-13
event handling, 9-17
  creating event handlers in specific
  ADEs, 9-18
DisplayExecution event, 9-19
DisplaySequenceFile event, 9-19
ExitApplication event, 9-18
ReportError event, 9-19
shutdown, 9-20
startup, 9-20
Wait event, 9-18
guidelines, 9-2
information source connections, 9-10
caption, 9-10
image, 9-11
numeric value, 9-12
interface pointer, obtaining, 9-17
LabVIEW, using with, 9-14
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
Operator Interface application, 9-13
  persistence of application settings, 9-30
startup and shutdown, 9-20
TSUtil Functions Library, 9-21
  view connections, 9-7
  visible controls (table), 9-5
Visual C++, using with, 9-16
Visual Studio, using with, 9-15
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
  localization functions, 9-26
  menus
    creating items, 9-23
    updating, 9-24
Thread Priority step, B-26
Threads pane, 3-6
Tools menu, 8-2
training and certification (NI resources), F-1
translators, 15-1
  deploying, 15-4
  DLLs, creating, 15-2
  examples, 15-2
  versioning, 15-3
troubleshooting (NI resources), F-1
type conflicts. See types
type definitions. See types
Index

type palette files, 11-2, 11-3, 11-8
restricting automatic type conflict
resolution with, 11-5
Type Properties dialog box, 12-8
types
See also built-in; data type; step type
conflicts, resolving, 11-3
creating, 11-1
definitions
modifying, 11-1
revisions, managing, 11-3
Allow Automatic Type Conflict
Resolution options
common scenarios for
(table), 11-7
avoiding unwanted propagation,
11-4, 11-5
execution, resolving types during
(note), 11-4
restricting automatic type conflict
resolution using type palette
files, 11-5
storing, 11-1
type palette files. See type palette files
Types pane, 2-2, 11-8
Types Window, 11-7
versioning, 11-2

See also connections; Manager
controls; TestStand User Interface
(UI) Controls; visible controls
deploying. See TestStand Deployment
Utility
distributing, 14-14
documenting custom, 9-32
defining user interfaces, 9-1
localization, 9-25
menus and menu items, 9-23
  updating, 9-24
starting up, 9-20
startup options
  configuring (table), 8-11
TestStand User Interface (UI) Controls,
1-6, 9-2
User Manager window, 11-9
User object, 7-3
user privileges. See privileges
utility subsequences
  Batch process model, A-27
  Parallel process model, A-13
  Sequential process model, A-9

V
variables
See also properties
expressions, using, 1-10
local, 2-3
  lifetime, 3-3
  monitoring, 3-2
sequence file global, 2-2
sequence local, 1-13
standard and custom data types, 1-11
station global, 1-9, 11-9
Variables pane, 2-2, 2-3, 2-4, 2-5, 3-6, 12-4

U
unit under test (UUT), 1-2
Use Auto Scheduled Resource step, B-25
user interface controls. See TestStand User
Interface (UI) Controls
user interfaces, 1-1, 1-4, 9-1
  application manifests, 9-34
  application styles, 9-26
  Authenticode signatures, 9-32
  creating, 9-1

NI TestStand Reference Manual I-18 ni.com
Variables
  View control
    description (table), 9-7
  versioning
    custom sequence files, 15-3
    translators, 15-3
    types, 11-2
  view connections, 9-7
  View Types For pane, 11-7
  VIs, processing with the TestStand Deployment Utility, 14-6
visible controls
  description of controls (table), 9-5
  Manager controls, connecting, 9-7
Visual C++. See Microsoft
Visual Studio .NET 2003. See Microsoft
Visual Studio. See Microsoft

W
Wait step, B-20
Watch View pane, 3-6
While step, 4-17
windows
  Execution, 3-3
  Sequence File, 2-4
  Sequence Hierarchy, 2-5
  Station Global, 11-9
  Types, 11-7
  User Manager, 11-9
Windows
  Authenticode signatures, 9-32
  remote execution, 5-20
  security, setting, 5-20
Windows 7
  application manifests, 9-34
  remote execution
    firewall settings, configuring, 5-26
    synchronization objects
      firewall settings, configuring, B-11
    TestStand, using with
directory structure, 8-2
Windows Vista
  application manifests, 9-34
  remote execution
    firewall settings, configuring, 5-27
    synchronization objects
    firewall settings, configuring, B-12
  TestStand, using with
directory structure, 8-2
Windows XP SP2
  remote execution
    firewall settings, configuring, 5-28
    synchronization objects
    firewall settings, configuring, B-13
    TestStand, using with
directory structure, 8-2
workspace files, 2-5
  creating for deployment, 14-3
dynamically called files, adding, 14-12
    options file, 2-6
    source code control (SCC), 2-6
tests, distributing, 14-11
Workspace pane, 2-5

X
XML reports
  associating style sheets, 6-19
displaying measurement data, 6-21
    Internet Explorer security settings
    affecting display, 6-22
distributing, 6-21
example transformation, 6-20
schema, 6-18
transforming with XSLT, 6-19