NATIONAL INSTRUMENTS™

TestStand™

TestStand User Manual

NATIONAL INSTRUMENTS™

December 1998 Edition
Part Number 322016A-01

Internet Support
E-mail: `support@natinst.com`
FTP Site: `ftp.natinst.com`
Web Address: `http://www.natinst.com`

Bulletin Board Support
BBS United States: 512 794 5422
BBS United Kingdom: 01635 551422
BBS France: 01 48 65 15 59

Fax-on-Demand Support
512 418 1111

Telephone Support (USA)
Tel: 512 795 8248
Fax: 512 794 5678

International Offices
Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 288 3336,
Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 725 725 11,
France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 6120092, Italy 02 413091,
Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 520 2635, Netherlands 0348 433466, Norway 32 84 84 00,
Singapore 2265886, Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200,
United Kingdom 01635 523545

National Instruments Corporate Headquarters
6504 Bridge Point Parkway Austin, Texas 78730-5039 USA Tel: 512 794 0100

© Copyright 1998 National Instruments Corporation. All rights reserved.

Important Information

Warranty

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

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

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

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

Copyright

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

Trademarks

CVI™, LabVIEW™, and TestStand™ are trademarks of National Instruments Corporation.

Product and company names listed are trademarks or trade names of their respective companies.

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

Contents

About This Manual

Organization of This Manual xxiii
Conventions Used in This Manual xxiv
Related Documentation xxv
Customer Communication xxv

Chapter 1
TestStand Architecture Overview

General Test Executive Concepts 1-1
TestStand Capabilities and Concepts 1-2
Major Software Components of TestStand 1-4
TestStand Sequence Editor 1-5
TestStand Run-Time Operator Interfaces 1-5
TestStand Test Executive Engine 1-6
Module Adapters 1-6
TestStand Building Blocks 1-7
Variables and Properties 1-7
Expressions 1-8
Categories of Properties 1-9
Steps 1-11
Built-In Step Properties 1-11
Step Types 1-12
Sequences 1-15
Sequence Parameters 1-15
Sequence Local Variables 1-15
Lifetime of Locals Variables, Parameters, and Custom Step Properties 1-15
Step Groups 1-16
Built-in Sequence Properties 1-16
Sequence Files 1-16
Storage of Types in Files 1-17
Process Models 1-17
Station Model 1-17
Main Sequence and Client Sequence File 1-18
Model Callbacks 1-18
Entry Points 1-19
Automatic Result Collection 1-22

Callback Sequences 1-23
Engine Callbacks 1-23
Front-End Callbacks 1-24
Sequence Executions 1-24
Normal and Interactive Executions 1-25
Terminating and Aborting Executions 1-26

Chapter 2
Sequence Editor Concepts

Sequence Editor Screen 2-1
Windows 2-2
Views 2-2
Tabs 2-3
Lists and Trees 2-3
Context Menus 2-4
Copy, Cut, and Paste 2-5
Drag and Drop 2-5
Menu Bar 2-5
Toolbars 2-5
Status Bar 2-5
Sequence Editor Windows 2-6
Sequence File Window 2-6
Execution Window 2-6
Type Palette Window 2-7
Station Globals Window 2-8
Users Window 2-9
Basics of Using TestStand 2-9
Creating a Sequence 2-9
Controlling Sequence Flow 2-13
Post Action 2-14
Preconditions 2-14
Goto Built-In Step Type 2-16
Run-Time Errors 2-16
Running a Sequence 2-16
Debugging a Sequence 2-17
Generating Test Reports 2-18
Using an Operator Interface 2-20

Chapter 3
Configuring and Customizing TestStand

Configuring TestStand 3-1
Sequence Editor Startup Options 3-1
Configure Menu 3-1
Customizing TestStand 3-3
TestStand Directory Structure 3-3
NI and User Subdirectories 3-4
The Components Directory 3-4
Creating String Resource Files 3-6
Resource String File Format 3-7
Using Data Types 3-8
Creating Step Types 3-8
Using the Tools Menu 3-9
Customizing the Engine and Front-End Callbacks 3-9
Modifying the Process Model 3-10
Using Process Model Callbacks 3-10
Creating Code Templates 3-11
Modifying Run-Time Operator Interfaces 3-11
Adding Users and Managing User Privileges 3-11

Chapter 4
Sequence Editor Menu Bar

Menus 4-1
File Menu 4-1
Login 4-2
Logout 4-2
New 4-2
Open 4-2
Close 4-2
Save 4-2
Save As 4-2
Save All 4-3
Unload All Modules 4-3
Most Recently Opened Files 4-3
Exit 4-3
Edit Menu 4-3
Cut and Copy 4-4
Paste 4-4
Delete 4-4
Select All 4-4
Sequence Properties 4-5

Sequence File Properties 4-5
Sequence File Callbacks 4-6
View Menu 4-7
Station Globals 4-8
Type Palette 4-8
User Manager 4-8
Paths 4-8
Find Type 4-11
Browse Sequence Context 4-12
Toolbars 4-13
Status Bar 4-13
Launch Report Viewer 4-13
Execute Menu 4-14
Execution Entry Point List 4-14
Run Active Sequence 4-14
Restart 4-14
Run Selected Steps 4-15
Loop on Selected Steps 4-15
Break On First Step 4-16
Tracing Enabled 4-16
Debug Menu 4-17
Resume 4-17
Step Over 4-17
Step Into 4-17
Step Out 4-17
Break 4-18
Terminate 4-18
Abort (no cleanup) 4-18
Break All 4-18
Terminate All 4-18
Abort All (no cleanup) 4-18
Resume All 4-18
Configure Menu 4-19
Station Options 4-19
Search Directories 4-29
External Viewers 4-30
Adapters 4-30
Report Options 4-30
Tools Menu 4-31
Sequence File Documentation 4-31
Sequence File Converters 4-31
Import/Export Limits 4-31
Update Automation Identifiers 4-31

Run Engine Installation Wizard 4-32
Customize 4-32
Window Menu 4-34
Cascade 4-34
Tile 4-34
Close Completed Execution Displays 4-34
Open Windows 4-34

Chapter 5
Sequence Files

Sequence File Window Views 5-1
All Sequences View 5-2
Sequence View Context Menu 5-3
Open Sequence 5-3
Insert Sequence 5-3
Rename 5-3
Browse Sequence Context 5-3
View Contents 5-3
Sequence Properties 5-3
Sequence File Properties 5-6
Sequence File Callbacks 5-9
Individual Sequence View 5-10
Main, Setup, and Cleanup Tabs 5-11
Step Group List View and Tree View 5-11
Step Group List View Columns 5-12
Step Group Context Menu 5-14
Parameters Tab 5-27
Parameters Tab Context Menu 5-27
Locals Tab 5-30
Locals Tab Context Menu 5-31
Preconditions Dialog Box 5-32
Sequence File Globals View 5-36
Lifetime and Scope of Sequence File Global Variables 5-36
Sequence File Globals View Context Menu 5-37
Insert Global 5-37
View Contents 5-38
Go Up One Level 5-38
Browse Sequence Context 5-38
Rename 5-38
Properties 5-38
Sequence File Types View 5-39

Chapter 6
Sequence Execution

Sequence Editor and Run-Time Operator Interfaces 6-1
What is an Execution? 6-1
Starting an Execution 6-2
Execution Entry Points 6-2
Executing a Sequence Directly 6-2
Interactively Executing Steps 6-3
Sequence Editor Execution Window 6-3
Steps Tab 6-4
Tracing 6-4
Debugging 6-5
Steps Tab Columns 6-5
Steps Tab Context Menu 6-6
Context Tab 6-7
Context Tab Context Menu 6-8
Report Tab 6-9
Call Stack Pane 6-10
Watch Expression Pane 6-12
Edit Expression 6-12
Add Watch 6-13
Modify Value 6-13
Refresh 6-13
Status Bar 6-13
Result Collection 6-14
Custom Result Properties 6-16
Standard Result Properties 6-17
Subsequence Results 6-18
Loop Results 6-19
Engine Callbacks 6-20
Step Execution 6-23
Step Status 6-24
Run-Time Errors 6-25

Chapter 7
Station Global Variables

Station Globals Window 7-1
Station Globals View Ring 7-2
Globals View Context Menu 7-2
Insert Global 7-2
View Contents 7-3
Go Up One Level 7-3

Browse Sequence Context 7-3
Rename 7-3
Global Variable Properties 7-3
Reload Station Globals 7-4
Persistence 7-4
Special TestStand Station Globals 7-5

Chapter 8
Sequence Context and Expressions

Sequence Context 8-1
Sequence Context Subproperties 8-3
StationGlobals 8-3
RunState 8-4
RunState.SequenceFile and Other SequenceFile Objects 8-8
RunState.Sequence and Other Sequence Objects 8-9
RunState.Step and Other Step Objects 8-10
RunState.InitialSelection 8-10
Using the Sequence Context 8-11
Expressions 8-12

Chapter 9
Types

Windows and Views that Display Types 9-1
Storage of Types in Files and Memory 9-2
Using Data Types 9-3
Specifying Array Sizes 9-5
Dynamic Array Sizing 9-7
Empty Arrays 9-7
Display of Data Types 9-8
Modifying Data Types and Values 9-9
Single Values 9-9
Arrays 9-11
Containers 9-11
Using the Standard Named Data Types 9-12
Path 9-12
Error and Common Results 9-13
Creating and Modifying Data Types 9-13
Custom Data Types Tab Tree and List Views 9-13
Value Field 9-16
Creating a New Custom Data Type 9-17
Adding Fields to Data Types 9-18

Properties Dialog Box for Custom Data Types 9-19
Property Dialog Box for Data Type Fields 9-20
Using Step Types 9-21
Creating and Modifying Custom Step Types 9-22
Custom Step Type Properties 9-23
Built-In Step Type Properties 9-24
General Tab 9-26
Menu Tab 9-28
Substeps Tab 9-30
Disable Properties Tab 9-32
Code Templates Tab 9-34
View Contents Button 9-41
Type Palette Window 9-41

Chapter 10
Built-In Step Types

Overview 10-1
Common Custom Properties 10-1
Step Status, Error Occurred Flag, and Run-Time Errors 10-2
Customizing Built-In Step Types 10-2
Step Types That You Can Use with Any Module Adapter 10-3
Action 10-3
Pass/Fail Test 10-4
Numeric Limit Test 10-6
String Value Test 10-9
Step Types That Work With a Specific Module Adapter 10-12
Sequence Call 10-12
Step Types That Do Not Use Module Adapters 10-14
Statement 10-14
Message Popup 10-15
Call Executable 10-18
Limit Loader 10-21
Import/Export Limits Command in the Tools Menu 10-26
Goto 10-28
Label 10-29

Chapter 11
User Management

User Manager Window 11-1
Users View 11-2
User List Tab 11-3
User List Context Menu 11-3

Profiles Tab 11-5
Profiles Tab Context Menu 11-6
Types View 11-7
User Standard Data Types 11-8
Adding New Properties and Privileges to the User Data Type 11-10
Verifying User Privileges 11-11
Accessing Privilege Settings for the Current User 11-11
Accessing Privilege Settings for Any User 11-12

Chapter 12
Module Adapters

Overview 12-1
Configuring Adapters 12-2
Source Code Templates 12-3
DLL Flexible Prototype Adapter 12-4
Configuring the DLL Adapter 12-4
Specifying a DLL Adapter Module 12-5
Module Tab 12-5
Source Code Tab 12-10
Debugging DLLs 12-12
Using MFC in a DLL 12-13
LabVIEW Standard Prototype Adapter 12-13
LabVIEW Standard Prototype Adapter Module Structure 12-13
Test Data Cluster 12-14
Error Out Cluster 12-16
Input Buffer 12-17
Invocation Information 12-17
Sequence Context 12-18
Configuring the LabVIEW Standard Prototype Adapter 12-19
Specifying a LabVIEW Standard Prototype Adapter Module 12-20
Debugging a LabVIEW Standard Prototype Adapter Module 12-21
C/CVI Standard Prototype Adapter 12-23
C/CVI Standard Adapter Module Prototypes 12-23
Example C/CVI Standard Prototype Code Module 12-27
Specifying a C/CVI Standard Prototype Adapter Module 12-28
Configuring the C/CVI Standard Prototype Adapter 12-31
Executing Code Modules In-Process 12-32
Executing Code Modules in an External Instance of LabWindows/CVI 12-34
Sequence Adapter 12-35
Specifying a Sequence Adapter Module 12-36
Edit Sequence Call Tab 12-37

Remote Execution Tab 12-39
Setting up TestStand as a Server for Remote Execution 12-41
ActiveX Automation Adapter 12-43
Configuring the ActiveX Automation Adapter 12-43
Specifying an ActiveX Automation Adapter Module 12-44
Running and Debugging ActiveX Automation Servers 12-49
Using ActiveX Servers with TestStand 12-49
Registering a Server 12-49
Compatibility Issues with Visual Basic 12-49

Chapter 13
Process Models

Directory Structure for Process Model Files 13-1
Special Editing Capabilities for Process Model Sequence Files 13-2
Sequence Properties Model Tab 13-3
Normal Sequences 13-3
Callback Sequences 13-4
Entry Point Sequences 13-4
Contents of the Default Process Model 13-8
Test UUTs Entry Point 13-12
Single Pass Entry Point 13-13
Support Files for the Default Process Model 13-14

Chapter 14
Managing Reports

Implementation of the Test Report Capability 14-1
Using Test Reports 14-2
Report Options Dialog Box 14-4
Contents Tab 14-5
Report File Pathname Tab 14-8

Chapter 15
Run-Time Operator Interfaces

Overview 15-1
TestStand Run-Time Operator Interfaces 15-2
The LabWindows/CVI Run-Time Operator Interface 15-2
The LabVIEW Run-Time Operator Interface 15-4
Building a Standalone Executable 15-4
The Visual Basic Run-Time Operator Interface 15-6
Distributing a Run-Time Operator Interface 15-8

Chapter 16
Distributing TestStand

Creating a Run-Time TestStand Engine Installation 16-1
Using a Custom TestStand Engine Installation 16-5
Distributing your Operator Interface 16-6
Installing the Customized Engine 16-6
LabVIEW 16-6
LabWindows/CVI 16-6
Visual Basic 16-7
Distributing Sequences and Code Modules 16-8
Distributing Sequence Files 16-8
Distributing DLL Code Modules 16-8
Distributing Object and Static Library Code Modules 16-8
Distributing LabVIEW Test VIs 16-9
Packaging VIs and SubVIs for a Sequence File 16-10
Distributing VIs by Saving Them without Full Hierarchy 16-10
Distributing VIs by Saving Them with Full Hierarchy 16-11
Distributing ActiveX Automation Code Modules 16-12
Customizing and Distributing a LabVIEW Run-Time Server 16-12
Rebuilding the TestStand LabVIEW Run-Time Server 16-13
Distributing the TestStand LabVIEW Run-Time Server 16-14

Appendix A
Customer Communication

Glossary

Index

Figures

Figure 1-1. TestStand System Architecture 1-4
Figure 1-2. The Expression Browser Dialog Box 1-9
Figure 1-3. Flowchart of TestUUTs Sequence in the Default Process Model 1-20
Figure 1-4. Test UUTs Entry Point Sequence in the Default TestStand Process Model 1-21
Figure 1-5. List of All Sequences in TestStand Process Model 1-22

Figure 2-1. Example Sequence Editor Screen 2-2
Figure 2-2. Example Sequence File Window 2-6
Figure 2-3. Example Execution Window 2-7

Figure 2-4. Example Type Palette Window 2-8
Figure 2-5. Example Station Globals Window 2-8
Figure 2-6. Example Users Window 2-9
Figure 2-7. Main Step Group in an Example Sequence 2-10
Figure 2-8. Insert Step Submenu 2-10
Figure 2-9. Step Properties Dialog Box 2-12
Figure 2-10. Preconditions Dialog Box 2-15
Figure 2-11. HTML Report for an Example Sequence. 2-19

Figure 4-1. File Menu 4-1
Figure 4-2. Edit Menu 4-3
Figure 4-3. Sequence Properties Dialog Box 4-5
Figure 4-4. Sequence File Properties Dialog Box 4-6
Figure 4-5. Sequence File Callbacks Dialog Box 4-7
Figure 4-6. View Menu 4-7
Figure 4-7. Edit Paths in Files Dialog Box 4-8
Figure 4-8. Edit Paths Dialog Box 4-9
Figure 4-9. Find Type Dialog Box 4-11
Figure 4-10. Browse Variables and Properties in Sequence Context Dialog Box 4-12
Figure 4-11. Execute Menu 4-14
Figure 4-12. Loop on Selected Steps Dialog Box—Loop Count Tab 4-15
Figure 4-13. Loop on Selected Steps Dialog Box—Stop Expression Tab 4-16
Figure 4-14. Debug Menu 4-17
Figure 4-15. Configure Menu 4-19
Figure 4-16. Execution Options 4-20
Figure 4-17. Time Limits Options 4-23
Figure 4-18. Preferences Options 4-25
Figure 4-19. Model Options 4-26
Figure 4-20. User Manager Options 4-27
Figure 4-21. Language Options 4-28
Figure 4-22. Search Directories Dialog Box 4-29
Figure 4-23. Tools Menu 4-31
Figure 4-24. Customize Tool Menu Dialog Box 4-32
Figure 4-25. Window Menu 4-34

Figure 5-1. Sequence File View Ring 5-2
Figure 5-2. All Sequences View in the Sequence File Window 5-2
Figure 5-3. Sequence Properties Dialog Box 5-4
Figure 5-4. General Tab on the Sequence File Properties Dialog Box 5-6
Figure 5-5. Advanced Tab on the Sequence File Properties Dialog Box 5-8
Figure 5-6. Callbacks Dialog Box 5-9
Figure 5-7. Individual Sequence View for an Example Sequence 5-10

Figure 5-8. The Step Group Tree View (Left) and List View (Right) 5-11
Figure 5-9. Step Group List View Columns for Steps 5-12
Figure 5-10. Step Group List View Columns for Step Properties 5-13
Figure 5-11. Insert Step Menu with LabVIEW Standard Prototype Adapter Selected 5-14
Figure 5-12. General Tab on the Step Properties Dialog Box 5-18
Figure 5-13. Run Options Tab on the Step Properties Dialog Box 5-19
Figure 5-14. Post Actions Tab on the Step Properties Dialog Box 5-22
Figure 5-15. Loop Options Tab on the Step Properties Dialog Box 5-24
Figure 5-16. Expressions Tab on the Step Properties Dialog Box 5-26
Figure 5-17. Parameters Tab 5-27
Figure 5-18. Insert Parameter Submenu 5-28
Figure 5-19. Locals Tab 5-30
Figure 5-20. Insert Local Submenu 5-31
Figure 5-21. Preconditions Dialog Box for a Sequence 5-33
Figure 5-22. Sequence File Globals View for an Example Sequence 5-36
Figure 5-23. Insert Global Submenu 5-37
Figure 5-24. Step Types Tab in Sequence File Types View 5-39

Figure 6-1. Steps Tab in the Sequence Editor Execution Window 6-4
Figure 6-2. The Context Tab in an Execution Window 6-8
Figure 6-3. HTML Report for an Example Sequence 6-10
Figure 6-4. Call Stack Pane while Suspended in a Subsequence 6-11
Figure 6-5. Steps Tab Displaying a Sequence Invocation in the Middle of the Call Stack 6-11
Figure 6-6. Watch Expression Pane 6-12
Figure 6-7. Execution Window Status Bar 6-13
Figure 6-8. A Result in a ResultList Array 6-15
Figure 6-9. Run-Time Error Dialog Box 6-26

Figure 7-1. Station Globals Window 7-1
Figure 7-2. The Insert Global Submenu 7-2

Figure 8-1. Variables/Properties Tab of the Expression Browser 8-13
Figure 8-2. Operators/Functions Tab of the Expression Browser 8-14

Figure 9-1. Type Conflict In File Dialog Box 9-3
Figure 9-2. Insert Local Submenu 9-5
Figure 9-3. Initial State of Array Bounds Dialog Box 9-5
Figure 9-4. Array Bounds Dialog Box with Settings for a Three-Dimensional Array 9-6
Figure 9-5. Array Bounds Dialog Box with an Initially Empty Array 9-7
Figure 9-6. Local Variables with Various Data Types 9-8

Figure 9-7. Properties Dialog Box for a Number Local Variable 9-10
Figure 9-8. Contents of Array Local Variable in List View 9-11
Figure 9-9. Standard Data Types Tab of the Type Palette Window 9-12
Figure 9-10. Custom Data Types Tab with Root Node Selected 9-14
Figure 9-11. Custom Data Types Tab Showing the Contents of a Container 9-15
Figure 9-12. Custom Data Types Tab Showing the Value Field for a Number 9-16
Figure 9-13. Modify Numeric Value Dialog Box 9-17
Figure 9-14. Insert Custom Data Type Submenu 9-17
Figure 9-15. Insert Fields Submenu 9-18
Figure 9-16. Properties Dialog Box for a Numeric Data Type 9-19
Figure 9-17. Insert Step Submenu 9-21
Figure 9-18. Step Types Tab of the Type Palette Window 9-23
Figure 9-19. Custom Properties of a Step Type 9-24
Figure 9-20. Step Type Properties Dialog Box—General Tab 9-26
Figure 9-21. Step Type Properties Dialog Box—Menu Tab 9-28
Figure 9-22. Step Type Properties Dialog Box—Substeps Tab 9-31
Figure 9-23. Step Type Properties Dialog Box—Disable Properties Tab 9-33
Figure 9-24. Step Type Properties Dialog Box—Code Templates Tab 9-37
Figure 9-25. Create Code Templates Dialog Box 9-38
Figure 9-26. Edit Code Template Dialog Box 9-39

Figure 10-1. Properties That All Steps Contain 10-1
Figure 10-2. Edit Pass/Fail Source Dialog Box 10-5
Figure 10-3. Pass/Fail Test Step Properties 10-5
Figure 10-4. Limits Tab on Edit Numeric Limit Test Dialog Box 10-6
Figure 10-5. Data Source Tab on Edit Numeric Limit Test Dialog Box 10-8
Figure 10-6. Numeric Limit Test Step Properties 10-8
Figure 10-7. Limits Tab on the Edit String Value Test Dialog Box 10-10
Figure 10-8. Data Source Tab on Edit String Value Test Dialog Box 10-11
Figure 10-9. String Limit Test Step Properties 10-11
Figure 10-10. Specify Module Dialog Box for Sequence Call Step 10-13
Figure 10-11. Edit Statement Step Dialog Box 10-15
Figure 10-12. Configure Message Box Step Dialog Box 10-16
Figure 10-13. Message Popup Step Properties 10-17
Figure 10-14. Configure Call Executable Dialog Box 10-19
Figure 10-15. Message Popup Step Properties 10-20
Figure 10-16. Example Sequence File with Limit Steps 10-22
Figure 10-17. Limits File Tab on Edit Limit Loader Step Dialog Box 10-22
Figure 10-18. Layout Tab on Edit Limit Loader Step Dialog Box 10-23
Figure 10-19. Limit Loader Step Properties 10-24
Figure 10-20. Import/Exports Sequence Limits Dialog Box 10-26
Figure 10-21. Edit Goto Step Dialog Box 10-28

Figure 11-1. Users View in the User Manager Window....11-2
Figure 11-2. User List Tab for Users View....11-3
Figure 11-3. Insert New User Dialog Box....11-4
Figure 11-4. Edit User Dialog Box....11-5
Figure 11-5. Profile Tab in the Users View....11-6
Figure 11-6. Types View in the User Manager Window....11-7
Figure 11-7. User Standard Data Type....11-8

Figure 12-1. Adapter Configuration Dialog Box....12-2
Figure 12-2. Choose Code Template Dialog Box....12-4
Figure 12-3. Specify Module Dialog Box for DLL Flexible Prototype Adapter—Module Tab....12-5
Figure 12-4. Specify Module Dialog Box for DLL Flexible Prototype Adapter—Source Code Tab....12-10
Figure 12-5. Test Data Cluster....12-14
Figure 12-6. Error Out Cluster....12-16
Figure 12-7. Invocation Information Cluster....12-17
Figure 12-8. Sequence Context Control....12-18
Figure 12-9. LabVIEW Adapter Configuration Dialog Box....12-19
Figure 12-10. Specify Module Dialog Box for LabVIEW Standard Prototype Adapter....12-20
Figure 12-11. Stepping into a LabVIEW VI....12-22
Figure 12-12. Specify Module Dialog Box for C/CVI Standard Prototype Adapter—Module Tab....12-29
Figure 12-13. Specify Module Dialog Box for C/CVI Standard Prototype Adapter—Source Code Tab....12-30
Figure 12-14. C/CVI Standard Adapter Configuration Dialog Box....12-32
Figure 12-15. Auto-Load Library Configuration Dialog Box....12-33
Figure 12-16. Example Sequence Parameters....12-36
Figure 12-17. Specify Module Dialog Box for the Sequence Adapter—Edit Sequence Call Tab....12-37
Figure 12-18. Specify Module Dialog Box for the Sequence Adapter—Remote Execution Tab....12-39
Figure 12-19. Specify Module Dialog Box for ActiveX Automation Adapter....12-44
Figure 12-20. Edit Parameter Value Dialog Box....12-47

Figure 13-1. Process Model Settings in the Advanced Tab of the Sequence File Dialog Box....13-2
Figure 13-2. Type Ring Control in the Sequence Properties Model Tab....13-3
Figure 13-3. Model Tab for an Execution Entry Point Sequence....13-5
Figure 13-4. List of All Sequences in the Default TestStand Process Model File....13-8

Figure 14-1. HTML Test Report in the Report Tab 14-3
Figure 14-2. ASCII Text Test Report in the Report Tab 14-4
Figure 14-3. Report Options Dialog Box—Contents Tab 14-5
Figure 14-4. Report Options Dialog Box—Report File Pathname Tab 14-8

Figure 16-1. Opening Dialog Box for the TestStand Engine Installation Wizard..... 16-1
Figure 16-2. Default Components to Include in the Installation 16-2
Figure 16-3. Customize Files to Include in Installation Dialog Box 16-3
Figure 16-4. Select Files to Include Dialog Box 16-4

Tables

Table 1-1. Callback Types..... 1-23

Table 2-1. Mouse and Keyboard Actions for Navigating List and Tree Views 2-3
Table 2-2. Standard Values for the Status Property after Execution Completes 2-13

Table 3-1. Sequence Editor Startup Options 3-1
Table 3-2. TestStand Subdirectories 3-3
Table 3-3. TestStand Component Subdirectories 3-5
Table 3-4. Resource String File Escape Codes 3-7

Table 6-1. Custom Properties in the Step Results for Steps That Use the Built-In Step Types 6-16
Table 6-2. Standard Step Result Properties 6-17
Table 6-3. Property Names for Subsequence Results 6-18
Table 6-4. Engine Callbacks 6-20
Table 6-5. Order of Actions That a Step Performs 6-23
Table 6-6. Standard Values for the Status Property 6-24

Table 8-1. First-Level Properties of the Sequence Context 8-2
Table 8-2. The StationGlobals TS Subproperty in the Sequence Context 8-3
Table 8-3. The RunState Subproperty in the Sequence Context 8-4
Table 8-4. The Subproperties of the SequenceFile Objects in the Sequence Context..... 8-8
Table 8-5. The Subproperties of the Sequence Objects in the Sequence Context..... 8-9
Table 8-6. The InitialSelection Subproperty in the Sequence Context 8-10
Table 8-7. Expression Operators 8-15
Table 8-8. Function Expression Operators 8-16
Table 8-9. Levels of Precedence in Expressions 8-19

Table 9-1. Adapter Dialog Box Names 9-22

Table 10-1. Numeric Limit Test Comparison Types 10-7

Table 11-1. Description of Subproperties in User Data Type 11-8

Table 12-1. TestStand Numeric Data Types 12-7
Table 12-2. TestStand String Data Types 12-8
Table 12-3. Adapter Interpretation of Ambiguous Declarations 12-12
Table 12-4. Test Data Cluster Elements 12-15
Table 12-5. Old Test Data Cluster Elements from LabVIEW Test Executive 12-16
Table 12-6. Error Out Cluster Elements 12-17
Table 12-7. Error Out Cluster Elements 12-18
Table 12-8. tTestData Structure Member Fields 12-23
Table 12-9. tTestError Structure Member Fields 12-26
Table 12-10. Step Properties Updated by C/CVI Standard Prototype Adapter 12-27
Table 12-11. Path Resolution of Sequence Pathnames for Remotely Executed Steps 12-40
Table 12-12. Variant Data Types Supported by the ActiveX Automation Adapter 12-48

Table 13-1. Order of Actions in the Test UUTs Entry Point 13-12
Table 13-2. Order of Actions in the Single Pass Entry Point 13-13
Table 13-3. Default Process Model Files 13-14

Table 15-1. Files in the LabWindows/CVI Run-Time Operator Interface Project File 15-2
Table 15-2. Top-Level Files in the LabVIEW Run-Time Operator Interface 15-4
Table 15-3. Top-Level Files in the Visual Basic Run-Time Operator Interface 15-6

Table 16-1. Custom TestStand Engine Installer Actions 16-5

About This Manual

Organization of This Manual

The *Product User* Manual is organized as follows:

- Chapter 1, *TestStand Architecture Overview*, describes the TestStand architecture and provides an overview of important TestStand concepts and components.
- Chapter 2, *Sequence Editor Concepts*, describes the various parts of the main window for the TestStand sequence editor. It also describes how you perform basic tasks in the sequence editor.
- Chapter 3, Configuring and Customizing TestStand, summarizes how you can configure and customize a TestStand station.
- Chapter 4, *Sequence Editor Menu Bar*, describes the menu items in the sequence editor menu bar.
- Chapter 5, *Sequence Files*, describes TestStand sequence files.
- Chapter 6, *Sequence Execution*, describes the execution of sequences in TestStand. It also describes the Execution window in the TestStand sequence editor.
- Chapter 7, *Station Global Variables*, describes station global variables and the Station Globals window.
- Chapter 8, *Sequence Context and Expressions*, describes the properties in the TestStand sequence context and how to use expressions in TestStand.
- Chapter 9, *Types*, discusses how you create, modify, and use step types, custom named data types, and standard named data types in TestStand. This chapter also describes the Type Palette window.
- Chapter 10, *Built-In Step Types*, describes the predefined step types that TestStand includes.
- Chapter 11, *User Management*, describes TestStand user management, the User Manager window, and how you can add users and manage user privileges.
- Chapter 12, *Module Adapters*, describes the module adapters that TestStand includes.
- Chapter 13, *Process Models*, discusses the default process model that TestStand includes. It also describes the directory structure that TestStand uses for process model files and the special capabilities that

the TestStand sequence editor has for editing process model sequence files.

- Chapter 14, *Managing Reports*, describes how you manage and use test reports in TestStand.
- Chapter 15, *Run-Time Operator Interfaces*, gives you an overview of how to create or customize an operator interface application. It also describes the various operator interface applications that TestStand includes.
- Chapter 16, *Distributing TestStand*, describes how to create an installer for a customized TestStand engine, how to distribute the TestStand engine with a run-time operator interface, and how to distribute each type of code module that TestStand supports. This chapter also describes how to customize and distribute a LabVIEW run-time server.
- Appendix A, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.
- The *Index* contains an alphabetical list of key terms and topics in this manual, including the page where you can find each one.

Conventions Used in This Manual

The following conventions are used in this manual:

<>

Angle brackets enclose the name of a key on the keyboard—for example, <Enter>.

[]

Square brackets enclose optional items—for example, [`response`].

-

A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <Ctrl-Alt-Delete>.

»

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

bold	Bold text denotes the names of menus, menu items, parameters, or dialog box buttons.
bold italic	Bold italic text denotes a note.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text from which you supply the appropriate word or value, as in Windows 3.*x*.
`monospace`	Text in this font denotes text or characters that you should literally 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, classes, entry points, properties, user profiles, login names, filenames and extensions, and for statements and comments taken from programs.
`monospace italic`	Italic text in this font denotes that you must enter the appropriate words or values in the place of these items.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, folders, and files.

Related Documentation

- *Getting Started with TestStand*
- *TestStand ActiveX API Reference* online help
- Ivo Salmre, “Building, Versioning, and Maintaining Visual Basic Components,” *Microsoft Developer Network*, Microsoft Corporation, February 1998.

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix A, *Customer Communication*, at the end of this manual.

1

TestStand Architecture Overview

This chapter describes the TestStand architecture and provides an overview of important TestStand concepts and components. This chapter introduces many terms and features that later chapters discuss in more detail. It is a good idea to become familiar with the contents of this chapter before proceeding to other chapters in the manual.

Getting Started with TestStand contains brief descriptions of TestStand components and the installation instructions for TestStand. It is a good idea to read *Getting Started with TestStand* before you read this manual. For a brief description of the TestStand sequence editor and how you perform basic tasks in it, refer to Chapter 2, *Sequence Editor Concepts*.

General Test Executive Concepts

A test executive is a program that allows you to organize and execute sequences of reusable test modules. The test modules often have a standard interface. Ideally, you can create the modules in a variety of programming environments.

This document uses a number of concepts that are applicable to test executives in general and some that are unique to the TestStand Test Executive. The following concepts are applicable to test executives in general.

- *Code module*—A program module, such as a Windows dynamic link library (`.dll`) or LabVIEW VI (`.vi`), containing one or more functions that perform a specific test or other action.
- *Test module*—A code module that performs a test.
- *Step*—Any action that you can include within a sequence of other actions, such calling a test module to perform a specific test.
- *Step module*—The code module that a step calls.

- *Sequence*—A series of steps you specify for execution in a particular order. Whether and when a step is executed can depend on the results of previous steps.
- *Subsequence*—A sequence that another sequence calls. You specify a subsequence call as a step in the calling sequence.
- *Sequence file*—A file that contains the definition of one or more sequences.
- *Sequence editor*—A program that provides a graphical user interface for creating, editing, and debugging sequences.
- *Run-time operator interface*—A program that provides a graphical user interface for executing sequences on a production station. A sequence editor and run-time operator interface can be separate application programs or different aspects of the same program.
- *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 run-time operator interface uses the services of a test executive engine.
- *Application Development Environment (ADE)*—A programming environment such as LabVIEW, LabWindows/CVI, or Microsoft Visual C, in which you can create test modules and run-time operator interfaces.
- *Unit Under Test (UUT)*—The device or component that you are testing.

TestStand Capabilities and Concepts

TestStand is a flexible, powerful test executive framework that has the following major features:

- Out-of-the-box configuration and components that give you a ready-to-run, full-featured test executive.
- Numerous ways for you to modify the out-of-the-box configuration and components or to add new components. These extensibility mechanisms enable you to create the test executive that meets your particular requirements without modifying the TestStand test execution engine. You can upgrade to newer versions of TestStand without losing your customizations.

- Sophisticated sequencing, execution, and debugging capabilities and a powerful sequence editor that is separate from the run-time operator interfaces.
- Three separate run-time operator interfaces with source code for LabVIEW, LabWindows/CVI, and Visual Basic.
- Independence from particular ADEs. You can create test modules in a wide variety of ADEs and call preexisting modules or executables. You can create your own run-time operator interface in any programming language that can control ActiveX automation servers.
- Conversion of sequence files from the LabVIEW Test Executive Toolkit Version 2.0 or the LabWindows/CVI Test Executive Toolkit Version 2.0 to TestStand.
- Comprehensive ActiveX API for building multithreaded test executives and other sequencing applications.

To provide these features, TestStand expands on the traditional test executive concepts and introduces many new ones. The new concepts include *step types*, *step properties*, *sequence variables*, *sequence parameters*, *module adapters*, and *process models*.

The remainder of this chapter consists of two major sections that introduce the new concepts as well as the enhancements to the traditional concepts. The first section discusses the major software components of TestStand. The second section discusses the features and building blocks in TestStand that you use to create test sequences and entire test systems.

Major Software Components of TestStand

This section provides an overview of the major software components of TestStand.

Figure 1-1 shows the high-level relationships between elements of the TestStand system architecture.

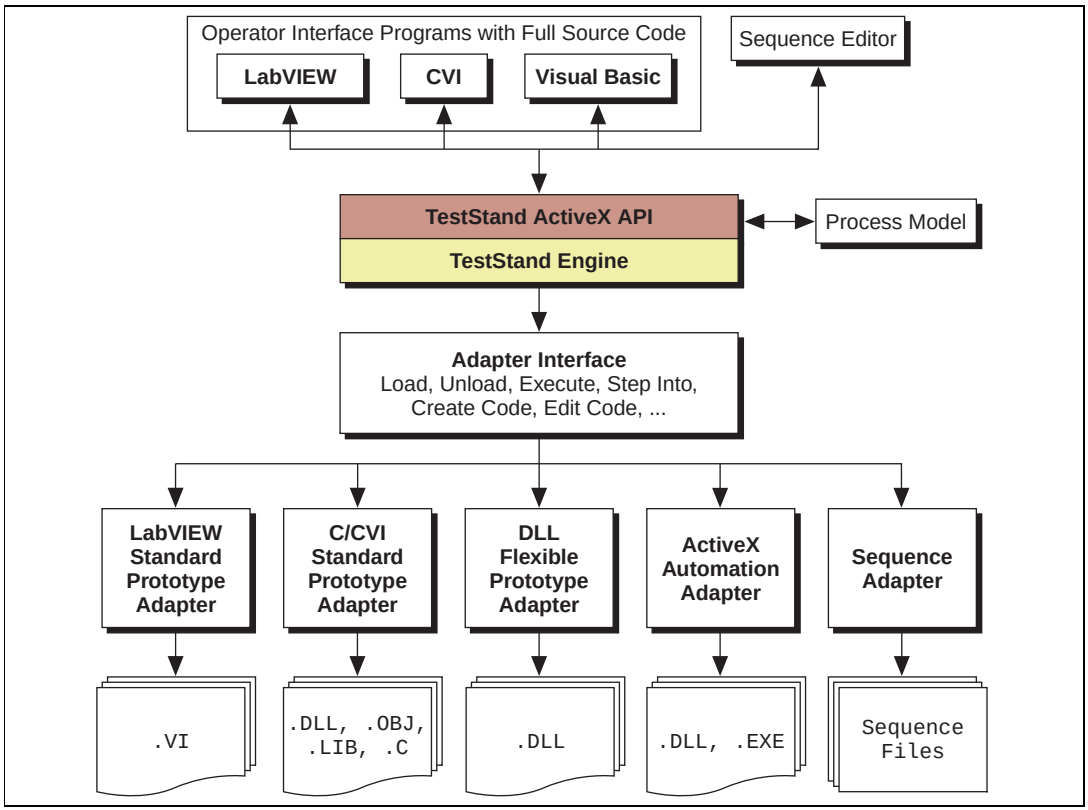

Figure 1-1. TestStand System Architecture

As shown in Figure 1-1, the TestStand engine plays a pivotal role in the TestStand architecture. The TestStand engine can run sequences. Sequences contain steps that can call external code modules. By using module adapters that have a standard adapter interface, the TestStand engine can load and execute different types of code modules. TestStand sequences can call subsequences through the same adapter interface. TestStand uses a special type of sequence called a process model to direct the high-level sequence flow. The TestStand engine exports an ActiveX

Automation API that the TestStand sequence editor and run-time operator interfaces use.

TestStand Sequence Editor

The TestStand sequence editor is an application program in which you create, modify, and debug sequences. The sequence editor gives you easy access to all the powerful TestStand features, such as step types and process models. The sequence editor has the debugging tools you are familiar with in ADEs such as LabVIEW, LabWindows/CVI, and Microsoft Visual C/C++. These debugging tools include breakpoints, single-stepping, stepping into or over function calls, tracing, a variable display, and a watch window.

In the TestStand sequence editor, you can start multiple concurrent executions. You can execute multiple instances of the same sequence, and you can execute different sequences at the same time. Each execution instance has its own Execution window. In trace mode, the Execution window displays the steps in the currently executing sequence. When execution is suspended, the Execution window displays the next step to execute and provides single-stepping options.

TestStand Run-Time Operator Interfaces

Your TestStand software includes three run-time operator interfaces in source and executable form. Each run-time operator interface is a separate application program. The operator interfaces differ primarily based on the programming language and ADE in which each is developed. TestStand ships with run-time operator interfaces developed in LabVIEW, LabWindows/CVI, and Visual Basic.

Although you can use the TestStand sequence editor at a production station, the TestStand run-time operator interfaces are simpler and fully customizable. Like the sequence editor, the run-time operator interfaces allow you to start multiple concurrent executions, set breakpoints, and single- step. Unlike the sequence editor, however, the run-time operator interfaces do not allow you to modify sequences, and they do not display sequence variables, sequence parameters, step properties, and so on.

If you want to customize one of the run-time operator interfaces, modify the source code for the program. If you want to write your own run-time operator interface, use the source code of one of the run-time operator interfaces as a starting point. Refer to Chapter 15, *Run-Time Operator Interfaces*, for more information on the run-time operator interfaces that ship with TestStand.

TestStand Test Executive Engine

The TestStand test executive engine is a set of DLLs that export an ActiveX Automation *Application Programming Interface* (*API*) you can use to create, edit, execute, and debug sequences. The TestStand sequence editor and run-time operator interfaces use the engine API. You can call the engine API from any programming environment that supports access to ActiveX Automation Servers. Thus, you can call the engine API from test modules, including test modules you write in LabVIEW and LabWindows/CVI.

The documentation for the engine API is available only as online help. You can access it through the **Help** menu of the sequence editor.

Module Adapters

Most steps in a TestStand sequence invoke code in another sequence or in a code module. When invoking code in a code module, TestStand must know the type of code module, how to call it, and how to pass parameters to it. The different types of code modules include LabVIEW VIs, objects in ActiveX Automation Servers, C functions in DLLs, and C functions in source, object, or library modules that you create in LabWindows/CVI or other compilers. Also, TestStand must know the list of parameters the code module requires.

TestStand uses *module adapters* to obtain this knowledge. TestStand currently provides the following module adapters for the following purposes:

- **DLL Flexible Prototype Adapter**—Calls C functions in a DLL with a variety of parameter types.
- **LabVIEW Standard Prototype Adapter**—Calls any LabVIEW VI that has the TestStand standard G parameter list.
- **C/CVI Standard Prototype Adapter**—Calls any C function that has the TestStand standard C parameter list. The function can be in an object file, library file, or DLL. It also can be in a source file that is in the project you are currently using in the LabWindows/CVI ADE.
- **ActiveX Automation Adapter**—Calls methods and accesses the properties of an ActiveX object.
- **Sequence Adapter**—Calls subsequences with parameters.

The module adapters contain other important information besides the calling convention and parameter lists. If the module adapter is specific to an ADE, the adapter knows how to open the ADE, how to create source

code for a new code module in the ADE, and how to display the source for an existing code module in the ADE. The ActiveX Automation Adapter and the DLL Flexible Prototype Adapter can query the type library for server or DLL for the parameter list information and display it to the sequence developer.

TestStand Building Blocks

This section provides an overview of the TestStand features and building blocks you use to create test sequences and entire test systems.

Variables and Properties

TestStand gives you various places in which you can store data values. These places are called *variables* and *properties*.

Variables are properties you can freely create in certain contexts. You can have variables that are *global* to a sequence file or *local* to a particular sequence. You also can have *station global* variables. The values of station global variables are persistent across different executions and even across different invocations of the sequence editor or run-time operator interfaces. The TestStand engine maintains the value of station global variables in a file on the run-time computer.

Each step in a sequence can have properties. For example, a step might have an integer error code property. The type of a step determines the set of properties it has. Refer to the *Step Types* section later in this chapter for more information on types of steps.

You can use TestStand variables to share data among tests that you write in different programming languages even if they do not have compatible data representations. You can pass values you store in variables and properties to code modules. You also can use the TestStand ActiveX API to access variable and property values directly from code modules.

When executing sequences, TestStand maintains a *sequence context* that contains references to all global variables and all local variables and step properties in active sequences. The contents of the sequence context changes depending on the currently executing sequence and step. If you pass a sequence context object reference to the code module, you can use the TestStand ActiveX API to access the variables and properties in the sequence context.

Expressions

In TestStand, 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. Sometimes you want to use an expression, which is a formula that calculates a new value from the values of multiple variable or properties. You can use an expression anywhere you can use a simple variable or property value. In expressions, you can access all variables and properties in the sequence context that is active when TestStand evaluates the expression. The following is an example of an expression:

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

TestStand supports all applicable expression operators and syntax that you use in C, C++, Java, and Visual Basic. If you are not familiar with expressions in these standard languages, TestStand also provides an expression browser dialog box you can access by clicking on the **Browse** button that appears next to controls that accept expressions. The expression browser allows you to interactively build an expression by selecting from lists of available variables, properties, and expression operators. The expression browser also lists a number of functions you can use in expressions. The expression browser has help text for each expression operator and function.

Figure 1-2 shows the Expression Browser dialog box.

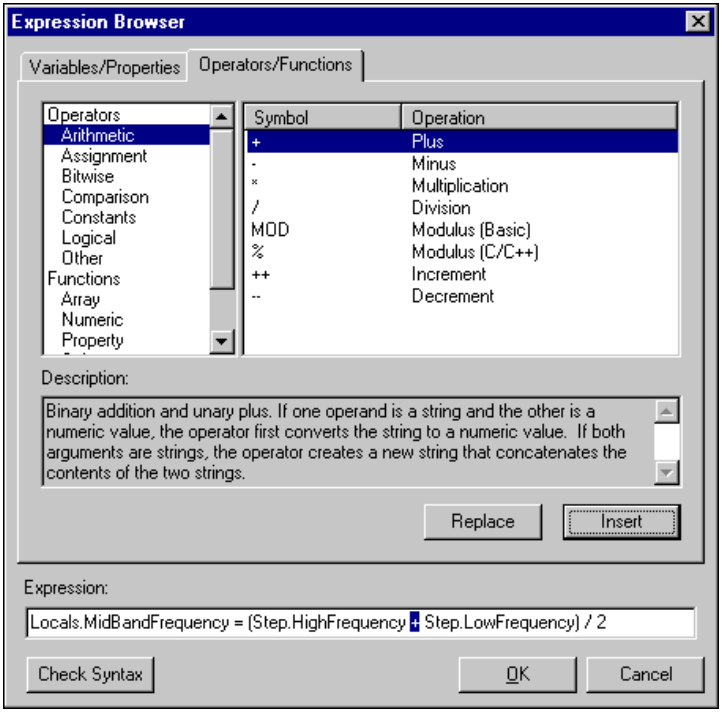

Figure 1-2. The Expression Browser Dialog Box

Categories of Properties

A property is a container of information. A property can contain a single value, an array of values for the same type, or no value at all. A property also can contain any number of subproperties. Each property has a name.

A value is a number, a string, a Boolean, or an ActiveX reference. TestStand stores numbers as 64-bit, floating-point values in the IEEE 754 format. TestStand stores an ActiveX reference as an `IDispatch` pointer or an `IUnknown` pointer. Values are not containers and thus cannot contain subproperties. Arrays of values can have multiple dimensions.

The following are the major categories of properties according the kinds of values they contain.

- A *single-valued property* contains a single value. Because TestStand has four types of values, TestStand has four types of single-valued properties: number properties, string properties, Boolean properties, and ActiveX reference properties.
- An *array property* contains an array of values. TestStand has number array properties, string array properties, Boolean array properties, and ActiveX reference array properties.
- A *property-array property* contains a value that is an array of subproperties of a single type. In addition to the array of subproperties, property-array properties can contain any number of subproperties of other types.
- A *container property* contains no values. Usually, container properties contain multiple subproperties. Container properties are analogous to structures in C/C++ and to clusters in LabVIEW.

Standard and Custom Named Data Types

When you create a variable or property, you specify its data type. In some cases, you use a simple data type such as a number or a Boolean. In other cases, you want to define your own data type in which you add subproperties to create an arbitrarily complex data structure. You can do so by creating a *named data type*. When you create a named data type, you can reuse it for multiple variables or properties. Although each variable or property you create with a named data type has the same data structure, the values they contain can differ.

TestStand defines certain *standard named data types*. You can add subproperties to the standard data types, but you cannot delete any of their built-in subproperties. The standard named data types are `Path`, `Error`, and `CommonResults`.

You can define your own *custom named data types*. You must choose a unique name for each of your custom data types. You can add or delete subproperties in each custom data type without restriction. For example, you might create a `Transmitter` data type that contains subproperties such as `NumChannels` and `PowerLevel`.

When you create a variable or property, you can select from among the simple property types and the named data types.

Built-In and Custom Properties

TestStand defines a number of properties that are always present for objects such as steps and sequences. An example is the step run mode property. TestStand normally hides these properties in the sequence editor, although it lets you modify some of them through dialog boxes. Such properties are called *built-in properties*.

You can define new properties in addition to the built-in properties. Examples are high- and low-limit properties in a step or local variables in a sequence. Such properties are called *custom properties*.

Steps

A sequence consists of a series of steps. In TestStand, a step can do many things, such as initializing an instrument, performing a complex test, or making a decision that affects the flow of execution in a sequence. Steps can perform these actions through several types of mechanisms. A step can jump to another step, execute an expression, call a subsequence, or call an external code module. This document refers to the code module that a step calls as the *step module*.

In TestStand, steps can have custom properties. For steps that call code modules, custom step properties are useful for storing parameters to pass to the code module for the step. They also serve as a place for the code module to store its results. You can use the TestStand ActiveX API to access the values of custom step properties from code modules.

Not all steps call code modules. Some steps perform standard actions you configure using a dialog box. In this case, custom step properties are useful for storing the configuration settings you specify.

Built-In Step Properties

TestStand steps have a number of built-in properties you can specify using the various tabs on the Step Properties dialog box. These built-in step properties include the following:

- *Preconditions* allow you to specify the conditions that must be true for TestStand to execute the step during the normal flow of execution in a sequence.
- *Load/Unload Options* allow you to control when TestStand loads and unloads the code modules or subsequences each step invokes.
- *Run Mode* allows you to skip a step or force it to pass or fail without executing the step module.

- *Record Results* allow you to specify whether TestStand stores the results of the step in a list. Refer to the *Automatic Result Collection* section later in this chapter for more information.
- *Step Failure Causes Sequence Failure* allows you to specify whether TestStand sets the status of the sequence to `Failed` when the status of the step is `Failed`.
- *Ignore Run-Time Errors* allows you to specify whether Test Stand continues execution normally after the step even though a run-time error occurs in the step.
- *Post Actions* allows you to execute callbacks or jump to other steps after executing the step, depending on the pass/fail status of the step or any custom condition.
- *Loop* options allow you to cause a single step to execute multiple times before executing the next step. You can specify the conditions under which to terminate the loop. You also can specify whether to collect results for each loop iteration, for the loop as a whole, or for both.
- *Pre Expressions* allow you to specify an expression to evaluate before executing the step module.
- *Post Expression*s allow you to specify an expression to evaluate after executing the step module.
- *Status Expression* allows you to specify an expression to use to set the value of the `status` property of the step automatically.

Step Types

Just as each variable or property has a data type, each step has a *step type*.

A step type can contain any number of custom properties. Each step of that type has the custom step properties in addition to the built-in step properties. All steps of the same type have the same properties, but the values of the properties can differ.

The step type specifies the initial values of all the step properties. When you create the step in the sequence editor, TestStand sets the initial values of the step properties from the values that the step type specifies.

You can modify the values of the built-in step properties by using the Step Properties dialog box. Usually, you can modify the values of custom step properties using a dialog box specific to the step type. If the step type does not have a dialog box for the custom properties, you can view the custom properties by selecting **View Contents** from the context menu for the step. Although step modules usually do not modify the values of the built-in step

properties at run time, they often modify and interrogate the values of the custom step properties.

A step type also can define standard behavior for each step of that type. It does this using a set of *substeps*. Substeps are actions that the TestStand engine performs for a step besides calling the step module. The substeps of a step type perform the same actions for every step of that type. The different types of substeps are as follows:

- Edit substep
- Pre Step substep
- Post Step substep

The sequence developer invokes the Edit substep by selecting a menu item in the context menu for the step or by clicking on a button in the Step Properties dialog box for the step. The step type specifies the name of the menu item and the caption of the button. The Edit substep displays a dialog box in which the sequence developer edits the values of custom step properties. For example, an Edit substep might display a dialog box in which the sequence developer specifies the high and low limits for a test. The Edit substep might then store the high and low limit values as step properties.

The engine calls the Pre Step substep before calling the step module. You can specify an adapter and a module to invoke in the Pre Step substep. For example, a Pre Step substep might call a code module that retrieves measurement configuration parameters and stores those parameters in step properties for use by the step module.

The engine calls the Post Step substep after calling the step module. You can specify an adapter and a module to invoke in the Post Step substep. A Post Step substep might call a code module that compares the values the step module stored in step properties against limit values the Edit substep stored in other step properties.

TestStand contains a set of predefined step types, as follows:

- Action
- Numeric Limit Test
- String Value Test
- Pass/Fail Test
- Label
- Goto
- Statement
- Limit Loader
- Message Popup
- Call Executable
- Sequence Call

For a description of each of these step types, refer to Chapter 10, *Built-In Step Types*. Although you can create a test application using only the predefined step types, you also can create your own step types. By creating your own step types, you can define standard, reusable classes of steps that apply specifically to your own application. For example, you might define a Switch Matrix Configuration step or a Transmitter Adjacent Channel Power Test step.

The sequence developer creates a new step by selecting the **Insert Step** item in the context menu that appears when you right-click on a sequence window. The **Insert Step** item opens a hierarchical submenu that contains the step types available on the computer. When you create a new step type, you specify its name and position within the submenu.

Source Code Templates

When you create a step type, you also can define *source code templates* for that step type. When the sequence developer creates a new step of that type, the developer can use a source code template to generate source code for the step module. For a particular step type, you can specify different source code templates for the different module adapters.

Sequences

In TestStand, a sequence consists of the following:

- Any number of local variables
- Any number of parameters
- A main group of steps
- A group of setup steps
- A group of cleanup steps
- Built-in sequence properties

Sequence Parameters

Each sequence has its own list of parameters. You can specify the number of parameters and the data type of each parameter. You also can specify a default value for each parameter. When the sequence developer creates a step that calls one sequence from another, the developer can specify the values to pass for the parameters of the subsequence. If the developer does not specify the value of a parameter, TestStand passes the default value. You can use the TestStand ActiveX API to access sequence parameter values from code modules that the steps in the sequence call.

You can pass local variables by value or by reference to any step in the sequence that calls a subsequence, a DLL using the DLL Flexible Prototype Adapter, or a method or property on an object using the ActiveX Automation Adapter.

Sequence Local Variables

You can create an unlimited number of local variables in a sequence. You can use local variables to store data relevant to the execution of the sequence. You can use the TestStand ActiveX API to access local variables from code modules that steps in the sequence call. You also can pass local variables by value or by reference to any step in the sequence that calls a subsequence, a DLL using the DLL Flexible Prototype Adapter, or a method or property on an object using the ActiveX Automation Adapter.

Lifetime of Locals Variables, Parameters, and Custom Step Properties

Multiple instances of a sequence can run at the same time. This can occur when you call a sequence recursively or when a sequence runs in multiple concurrent executions. Each instance of the sequence has its own 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.

Step Groups

A sequence can contain the following groups of steps: Setup, Main, and Cleanup. When TestStand executes a sequence, the steps in the Setup group execute first. The steps in the Main group execute next. The steps in the Cleanup group execute last. Usually, the Setup group contains steps that initialize instruments, fixtures, or a UUT. The Main group usually contains the bulk of the steps in a sequence, including the steps that test the UUT. The Cleanup group contains steps that power down or de-initialize the instruments, fixtures, and UUT.

One of the reasons for having separate step groups is to ensure that the steps in the Cleanup group execute regardless of whether the sequence completes successfully or a run-time error occurs in the sequence. If a Setup or Main step causes a run-time error to occur, the flow of execution jumps to the Cleanup step group. The Cleanup steps always run even if some of the Setup steps do not run. If a Cleanup step causes a run-time error, execution continues at the next Cleanup step.

If a run-time error occurs in a sequence, TestStand reports the run-time error to the calling sequence. Execution in the calling sequence jumps to the Cleanup group in the calling sequence. This process continues up through the top-level sequence. Thus, when a run-time error occurs, TestStand terminates execution after running all the Cleanup steps of the sequences that are active when the run-time error occurs.

Built-in Sequence Properties

Sequences have a few built-in properties that 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 property of the sequence to `Failed`.

Sequence Files

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

Sequences files have a few built-in properties you can specify using the Sequence File Properties dialog box. 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.

Storage of Types in Files

Each sequence file contains the definitions of all property and step types that the variables, parameters, and steps in the sequence file use. This is true for all TestStand files that use types.

In memory, TestStand allows only one definition for each type. If you load a file that contains a type definition and a type definition of the same name already exists in memory, TestStand verifies that the two type definitions are identical. If they are not identical, TestStand informs you of the conflict. You can select one of the definitions to replace the other, or you can rename one of them so that they can coexist.

Process Models

Testing a Unit Under Test (UUT) requires more than just executing a set of tests. Usually, the test executive must perform a series of operations before and after it executes the sequence that performs the tests. Common operations include identifying the UUT, notifying the operator of pass/fail status, generating a test report, and logging results. These operations define the testing *process*. The set of such operations and their flow of execution is called a *process model*. Some traditional test executives implement their process models internally and do not allow you to modify them. Other test executives do not define a process model at all. TestStand comes with a default process model that you can modify or replace.

Having a process model is essential so that you can write different test sequences without repeating standard testing operations in each sequence. Ability to modify the process model is essential because the testing process can vary based on your production line, your production site, or the systems and practices of your company.

TestStand provides a mechanism for defining a process model. A process model is in the form of a sequence file. You can edit a process model just as you edit your other sequences. TestStand ships with a fully functional default process model. You can write your own process model, or you can copy the default process model and then modify it.

Station Model

You can select a process model file to use for all sequence files. This process model file is called the *station model* file. The TestStand installation program establishes `TestStandModel.seq` as the station model file. You can use the Station Options dialog box to select a different station model. You also can use the Station Options dialog box to allow

individual sequence files to specify their own process model file, but usually this is not necessary.

Main Sequence and Client Sequence File

In TestStand, the sequence that initiates the tests on a UUT is called the *main sequence*. You must name each main sequence `MainSequence`. When you create a new sequence file, TestStand automatically inserts a `MainSequence` sequence in the file. The process model invokes the main sequence as part of the overall testing process. The process model defines what is constant about your testing process, whereas main sequences define the steps that are unique to the different types of tests you run.

When you begin an execution, you usually do so from a main sequence in one of your sequence files. 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 set the Station Options to allow sequence files to override your station model setting.

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

Model Callbacks

By default, each main sequence you execute uses the process model that you select for the entire test station. TestStand has a mechanism called a *model callback* that allows the sequence developer to customize the behavior of a process model for each main sequence that uses it. By defining one or more model callbacks in a process model, you specify the set of process model operations that the sequence developer can customize.

You define a model callback by adding a sequence to the process model file, marking it as a callback, and calling it from the process model. The sequence developer can override the callback in the model sequence file by using the Sequence File Callbacks dialog box to create a sequence of the same name in the client sequence file.

For example, the default TestStand process model defines a `TestReport` callback that generates the test report for each UUT. Normally, the `TestReport` callback in the default process model file is sufficient because it handles many types of test results. The sequence developer can, however, override the default `TestReport` callback by defining a different `TestReport` callback in a particular client sequence file.

Process models use callbacks to invoke the main sequence in the client sequence file. Each client sequence file must define a sequence by the name of `MainSequence`. The process model contains a `MainSequence` callback that is merely a placeholder. The `MainSequence` in the client sequence file overrides the `MainSequence` placeholder in the model file.

To alter the behavior of the process model for all sequences, you can modify the process model or replace it entirely. To redefine the set of customizable operations, you can define new callbacks in, or delete existing callbacks from, the process model file.

Entry Points

A process model defines a set of *entry points*. Each entry point is a sequence in the process model file. You mark a sequence in the model file as an entry point in the Sequence Properties dialog box.

By defining multiple entry points in a process model, you give the test station operator different ways to invoke a main sequence. For example, the default TestStand process model provides two entry points: `Test UUTs` and `Single Pass`. The `Test UUTs` entry point initiates a loop that repeatedly identifies and tests UUTs. The `Single Pass` entry point tests a single UUT without identifying it. Such entry points are called *execution entry points*. Execution entry points appear in the **Execute** menu of the sequence editor or operator interface when the active window contains a non-model sequence file that has a `MainSequence` callback.

Figure 1-3 contains a flowchart of the major operations of the `Test UUTs` entry point sequence in the default process model. Notice that the sequence implements many of its operations as callbacks. The box on the left shows

the flow of control. The box on the right shows the action that each callback in the default model performs if you do not override it.

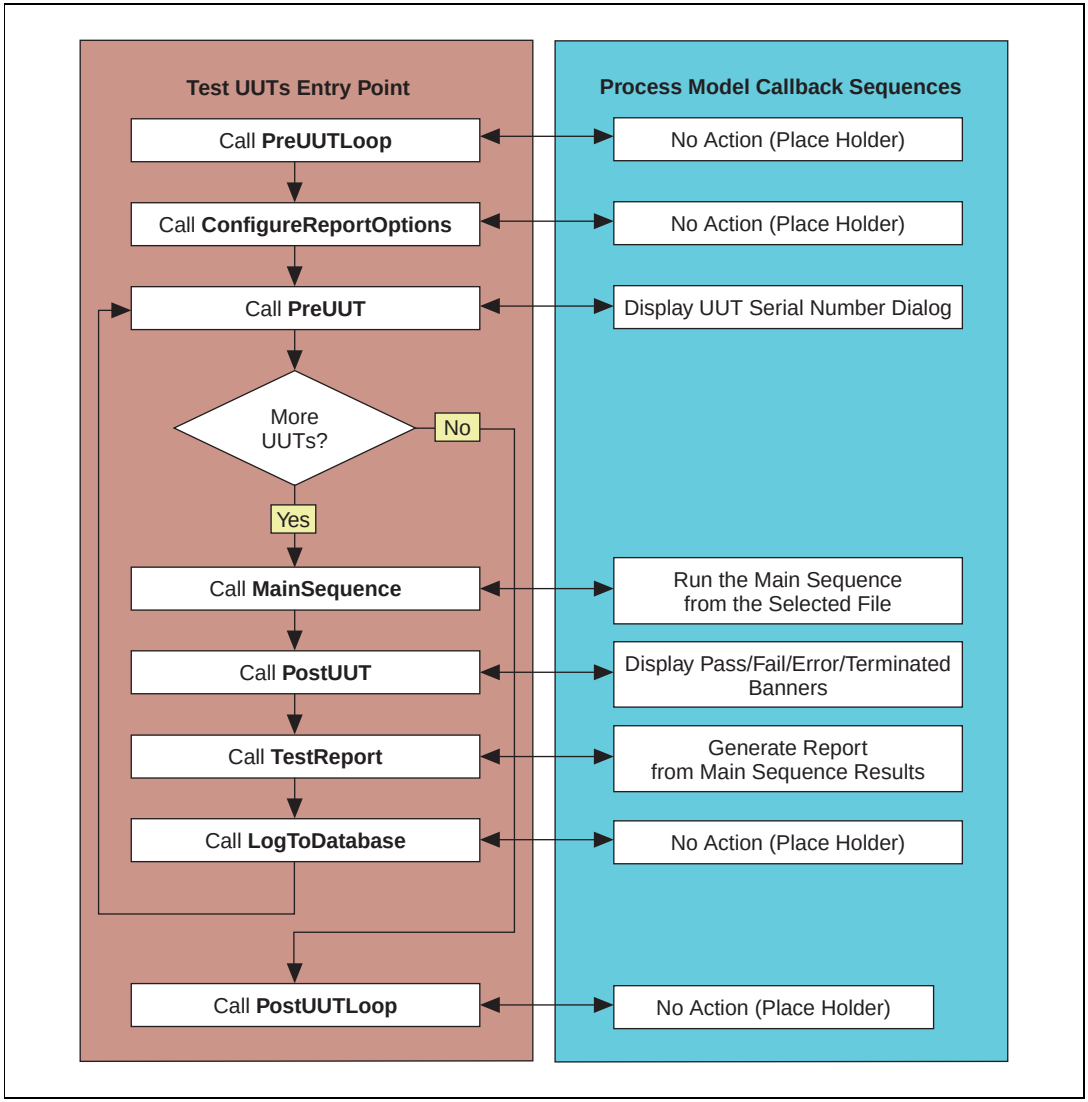

Figure 1-3. Flowchart of TestUUTs Sequence in the Default Process Model

Like any other sequence, the sequence for a process model entry point can contain calls to DLLs, calls to subsequences, Goto steps, and so on.

Figure 1-4 shows the entire set of steps for the `Test UUTs` entry point in the default process model.

TestStandModel.seq

Main | Setup | Cleanup | Parameters | Locals View: Test UUTs

Step	Description	Execution Flow	Comment
Check For Proper Use	Error	Pre, Post	In case someone accidently tries to execute ...
Clear Report	Action, SetReport(RunState.Execution, "");		Clear the report in case we are restarting a p...
PreUUTLoop Callback	Call PreUUTLoop (<Current File>)		
Get Report Options	Call Get Report Options (<Current File>)		Reads report options, and calls the ReportO...
Get Database Options	Call Get Database Options (<Current File>)	Pre	
Include Limits in Results	Action, AddExtraResult(RunState.Execution, "S...	Pre	Configures the execution to include limits in t...
Include Comparison Type in Results	Action, AddExtraResult(RunState.Execution, "S...	Pre	Configures the execution to include limit com...
Next UUT			
Increment UUT Index	Locals.UUT.UUTLoopIndex++		Increment the UUT index. The first UUT ind...
PreUUT Callback	Call PreUUT (<Current File>)		Display the UUT dialog box.
Goto End of UUT Loop If No More UUTs	Goto 'End of UUT Loop'	Pre, Post	
Determine Report File Path	Action, DetermineReportFilePathName(ThisCon...	Pre	Determines the report file path using the Rep...
Put 'Test In Progress' In Report	Action, SetReport(RunState.Execution, ResStr(...		Put the "Test in Progress" string into the rep...
Clear Results List	SetNumElements(Locals.ResultList, 0)		Discard results from previous loop.
Clear Report Local Variable	Locals.Report = ""		Discard the report from the previous loop.
Get Start Time	Locals.StartTime = Time(), Locals.StartDate = D...		
MainSequence Callback	Call MainSequence (<Current File>)		Call MainSequence in the client sequence file.
PostUUT Callback	Call PostUUT (<Current File>)		
Clear Report	Action, SetReport(RunState.Execution, "");		Remove the "Testing in progress" message.
TestReport Callback	Call TestReport (<Current File>)	Pre	Generates the test report string for the curre...
SetReport	Action, SetReport(RunState.Execution, Locals....		Attaches the report string for the current UU...
Write UUT Report	Action, WriteReport(RunState.Report, Locals.R...	Pre	Writes or appends the report string for the cu...
Log To Database Callback	Call Log To Database (Database.seq)	Pre	
Handle Termination	Testing Terminated for Current UUT	Pre, Post	If you terminate execution in the MainSeque...
Goto Next UUT	Goto 'Next UUT'	Post	
End of UUT Loop			
PostUUTLoop Callback	Call PostUUTLoop (<Current File>)		
CloseDBHandles	Call Close DB Handles (Database.seq)	Pre	
Read Entire Report	Action, ReadReport(RunState.Report, Locals.R...	Pre	Read entire report file so that the user can s...

Figure 1-4. Test UUTs Entry Point Sequence in the Default TestStand Process Model

You can execute a sequence without a process model by selecting the **Run *Sequence Name*** item in the **Execute** menu, where ***Sequence Name*** is the name of the sequence you are currently viewing. This option is useful for debugging. It executes the sequence directly, skipping the process model operations such as UUT identification and test report generation. You can execute any sequence this way, not just main sequences.

A process model can define other types of entry points, such as *configuration entry points*. An example is the `Config Report Options` entry point, which appears as **Report Options** in the **Configure** menu of the sequence editor or run-time operator interface. Refer to Chapter 13, *Process Models*, for more information on process model entry points.

Figure 1-5 shows a list of all the sequences in the default TestStand process model. The first three sequences are entry points. The last sequence is a utility subsequence that the execution entry points call. The other sequences are callbacks that you can override in a client sequence file.

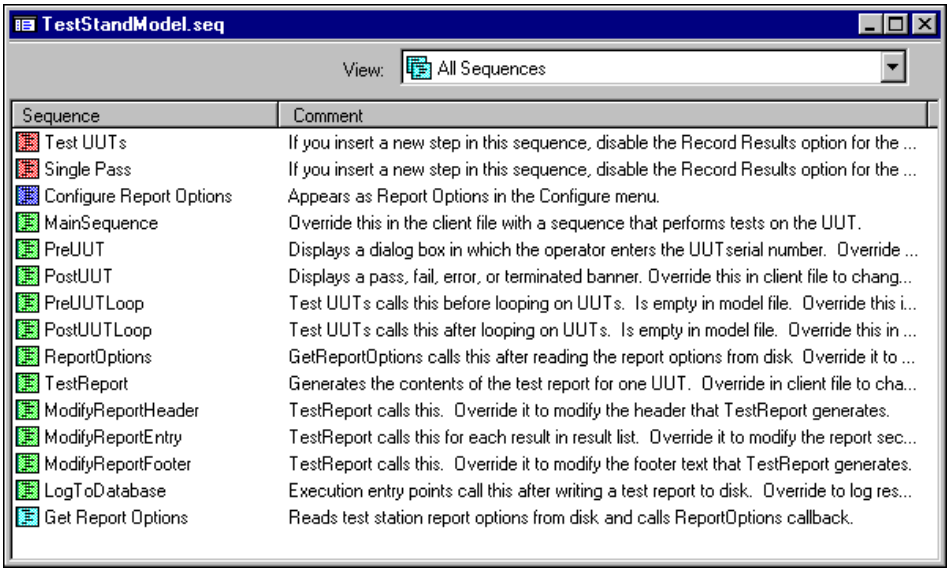

Figure 1-5. List of All Sequences in TestStand Process Model

Automatic Result Collection

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

Each sequence has a local array that stores the results of each step. The contents in the results for each step can vary depending on the step type. When TestStand stores the results for a step into the array, it 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 in Chapter 6, *Sequence Execution*, for more information on how TestStand collects results.

Callback Sequences

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

TestStand defines three categories of callbacks. The categories are based on the entity that invokes the callback and the location in which you define the callback. Table 1-1 shows the different types of callbacks.

Table 1-1. Callback Types

Callback Type	Where You Define the Callback	Who Calls the Callback
Model Callbacks	Process model file or client sequence file	Sequences in the process model file
Engine Callbacks	`StationCallbacks.seq`, the process model file, or a regular sequence file	Engine
Front-End Callbacks	`FrontEndCallbacks.seq`	Operator interface program

The *Process Models* section earlier in this chapter discusses model callbacks in detail.

Engine Callbacks

The TestStand engine defines a set of callbacks it invokes at specific points during execution. These callbacks are called *engine callbacks*. TestStand defines the name of each engine callback.

Engine callbacks are a way for you to configure TestStand to 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.

The engine callbacks are in three general groups, based on the file in which the callback sequence appears. You can define engine callbacks in sequence files, in process model files, and in the `StationCallbacks.seq` file.

Note ***TestStand installs an empty*** `StationCallbacks.seq` ***file in the*** `TestStand\Components\NI\Callbacks\Station` ***directory. You can add your own station engine callbacks in the*** `StationCallbacks.seq` ***file in the*** `TestStand\Components\User\Callbacks\Station` ***directory.***

Front-End Callbacks

Front-end callbacks are sequences in the `FrontEndCallbacks.seq` file that operator interface programs call. Front-end callbacks allow multiple operator interfaces to share the same implementation for a specific operation. The version of `FrontEndCallback.seq` that TestStand installs contains one front-end callback sequence, `LoginLogout`. The sequence editor and all operator interfaces that come with TestStand call `LoginLogout`.

When you implement operations as front-end callbacks, you write them as sequences. Thus you can modify a front-end callback without modifying the source code for the operator interfaces or rebuilding the executables for them. For example, to change how the various operator interfaces perform the login procedure, you only have to modify the `LoginLogout` sequence in `FrontEndCallbacks.seq`.

You can create new front-end callbacks by adding sequences to `FrontEndCallbacks.seq` file. You can then invoke this sequence from each of the operator interface programs you use. You invoke the sequence using functions in the TestStand ActiveX API. You cannot edit the source for the TestStand sequence editor. Thus, you cannot make the sequence editor call new front-end callbacks that you create.

Note ***TestStand installs predefined front-end callbacks in the*** `FrontEndCallbacks.seq` ***file in the*** `TestStand\Components\NI\Callbacks\FrontEnd` ***directory. You can add your own front-end callbacks or override a predefined callback in the*** `FrontEndCallbacks.seq` ***file in the*** `TestStand\Components\User\Callbacks\FrontEnd` ***directory.***

Sequence Executions

When you run a sequence, TestStand creates an *execution object*. The execution object contains all the information that TestStand needs to run your 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. Each execution runs in a different thread.

Usually, the TestStand sequence editor creates a new window for each execution. This window is called an *Execution window.* In the Execution window, you can view steps as they execute, the values of variables and properties, and the test report. Usually, run-time operator interface programs also have a view or window for each execution.

Normal and Interactive Executions

You can start an execution in the sequence editor by selecting the **Run *Sequence Name*** item or one of the process model entry points from the **Execute** menu. This is called a *normal execution*.

You can run steps in *interactive mode* by selecting one or more steps in and choosing the **Run Selected Steps** or **Loop Selected Steps** items in the *context menu*. In interactive mode, only the selected steps in the sequence execute, regardless of any branching logic that the sequence contains. The selected steps run in the order in which they appear in the sequence.

You can run steps in interactive mode from two different contexts. You run steps interactively from a Sequence File window. When you do so, you create a new execution. This is 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. Root interactive executions do not invoke process models. Thus, by default, root interactive executions do not generate test reports.

You also can run steps interactively from an existing Execution window for a normal execution that is suspended at a breakpoint. You can run steps only in the sequence and step group in which execution is suspended. When you do this, the selected steps run within the context of the normal execution. This is called a *nested interactive execution.* The steps that you run interactively can access the variable values of the normal execution and add to its results. When the selected steps complete, the execution returns to the step at which it was suspended when you chose **Run Selected Steps** or **Loop Selected Steps**.

Terminating and Aborting Executions

The menus in the sequence editor and run-time operator interfaces have commands that allow you to stop execution before the execution has completed normally. The TestStand engine API has corresponding methods that allow you to stop execution from a code module. You can stop one execution or all executions. You can issue the stop request at any time, but it does not take effect in each execution until the currently executing code module returns control.

You can stop executions in two ways. When you *terminate* an execution, all the Cleanup step groups in the sequences on the call stack run before execution ends. Also, the process model can continue to run. Depending on the process model, it might continue testing with the next UUT or generate a test report.

When you *abort* an execution, the Cleanup step groups do *not* run, and process model cannot continue. In general, it is better to terminate execution so that the Cleanup step groups can return your system to a known state. You abort an execution when you want the execution to stop completely as soon as possible. Usually, you abort an execution only when you are debugging and you are sure that is safe to not run the cleanup steps for a sequence.

2

Sequence Editor Concepts

This chapter describes the various parts of the main window for the TestStand sequence editor. It also describes how you perform basic tasks in the sequence editor.

Sequence Editor Screen

The sequence editor main window contains standard window features common to Windows applications, such as windows, menus, toolbars, and a status bar.

Figure 2-1 shows an example of the sequence editor main window.

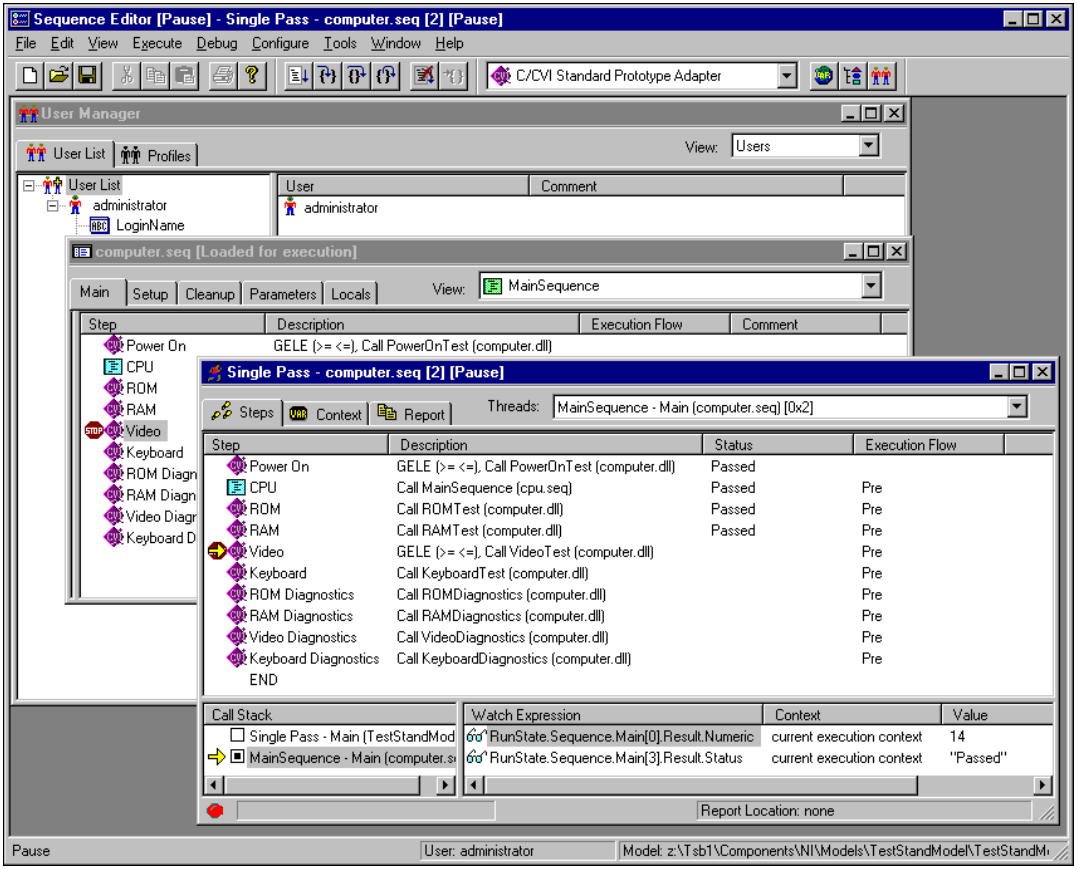

Figure 2-1. Example Sequence Editor Screen

Windows

The sequence editor uses child windows to display sequence files, sequence executions, station globals, data types, step types, users, and user privileges. This manual refers to these child windows simply as *windows*.

Views

Each TestStand window can contain different views to display various elements of sequence files, sequence executions, types, or globals. For example, a sequence file contains multiple sequences. The pull-down ring in the upper right corner of the Sequence File window selects the current view for the Sequence File window. The Sequence File window views

include a list of all sequences, a list of steps in a particular sequence, a list of the sequence file global variables, and a list of types that the sequence file uses.

Tabs

TestStand windows use a series of tabs to display detailed information unique to the view. For example, when viewing a sequence in a sequence file, the following tabs are available:

- **Main**—Displays the steps in the main sequence.
- **Setup**—Displays steps that execute before the Main step group runs.
- **Cleanup**—Displays steps that execute after the Main step group runs.
- **Parameters**—Lists the values that the sequence receives when another sequence calls it.
- **Locals**—Displays variables accessible by any of the steps in the sequence.

Lists and Trees

The sequence editor uses lists and trees to show the relationship and hierarchical nature of the data that appears in each view. For example, a list view for the Sequence File window displays all the sequences in a sequence file. When you select a sequence in that list and press <Ctrl-Enter>, the view changes to a list of all steps in the selected sequence. You also can display the steps of a sequence in a tree view where the steps are the nodes and the step properties are the branches of the tree.

Table 2-1 describes the standard behavior for keyboard and mouse actions that you perform on objects in list views and tree views.

Table 2-1. Mouse and Keyboard Actions for Navigating List and Tree Views

Mouse Action	Keyboard Action	Type of View	Behavior
Double-click	Press <Enter>	List view	Displays the Properties dialog box for the object.
<Ctrl>-Double-click	Press <Ctrl-Enter>	List view	Expands the object to show its contents.
Double-click on a closed node	Press <Enter> or <+>	Tree view	Expands the tree view node.
Double-click on an opened node	Press <Enter> or <->	Tree view	Collapses the tree view node.

Table 2-1. Mouse and Keyboard Actions for Navigating List and Tree Views (Continued)

Mouse Action	Keyboard Action	Type of View	Behavior
Click to select node	Use arrows to select node	Tree view	Show contents of tree node in list view
<Alt>-Double-click	Press <Alt-Enter>	List or tree view	Displays the Properties dialog box for the object.
(None)	<Backspace>	List or tree view	Go up one level in tree view and show contents of that level in list view

Each item in a list view can have multiple columns. For example, a step in a list view has a Step column, a Description column, an Execution Flow column, and a Comment column. You can expand a column to the width of its largest entry by double-clicking on the vertical separator at the right edge of the column heading. This is especially useful when an item has a long comment.

Context Menus

You can open a context menu in a window by pressing the right mouse button. The list of menu items in a context menu varies depending on the view, the mouse position, and whether any items are selected. Most of the context menu items do not appear in the main window menu bar, so you can access them only from the context menus. For example, you can insert a step into a sequence only by using the **Insert Step** context menu item.

Certain items appear in several different context menus. They act upon the object on which you right-click to display the context menu. For example, the **Properties** item displays the Properties dialog box for the object. The **View Contents** item displays the contents of the object in the list view.

In the context menu you can display by right-clicking on the list view background, the **Go Up 1 Level** menu item moves up one level in the tree view and shows the contents of that level in the list view.

Copy, Cut, and Paste

When displaying sequences, steps, types, or globals in a list or tree format, you can cut, copy, and paste items between different views and windows.

Drag and Drop

When displaying sequences, steps, types, or globals in a list or tree format, you can drag and drop items between different views and windows.

Menu Bar

The sequence editor uses a common menu bar. Some menu items may be dim depending on the state of the sequence editor session and which window is active. Refer to Chapter 4, *Sequence Editor Menu Bar*, for information on the sequence editor main menu bar and menu items.

Toolbars

Toolbars and their icons give you quick access to commonly used menu items. To find out what a toolbar button does, position the mouse cursor over the button. A help description appears on the status bar of the main window.

The sequence editor maintains three toolbars: the Standard, Debug, and Environment toolbars. To configure which toolbars are visible, select **View»Toolbar** or right-click on the toolbar area.

Status Bar

The status bar at the bottom of the sequence editor window displays the current state of the editor or displays help information. The left portion of the status bar displays help messages. When you select menu items or toolbar icons, a short description appears for the selected item. Otherwise, the status bar displays the current execution state, such as `edit` or `running`.

The center portion of the status bar displays the current user and the process model for the active sequence window.

The right portion of the status bar displays the state of the keyboard, such as numeric lock indicator.

Sequence Editor Windows

This section describes all the windows in the sequence editor.

Sequence File Window

In the sequence editor, you use a Sequence File window to view and edit a sequence file. Figure 2-2 shows an example Sequence File window. You can use the View ring control at the top right of the Sequence File window to view an individual sequence, a list of all sequences in the file, the global variables in the file, or the types you use in the file. In an individual sequence view, you use the tabs to view the Main, Setup, or Cleanup step groups, the sequence parameters, or the sequence local variables.

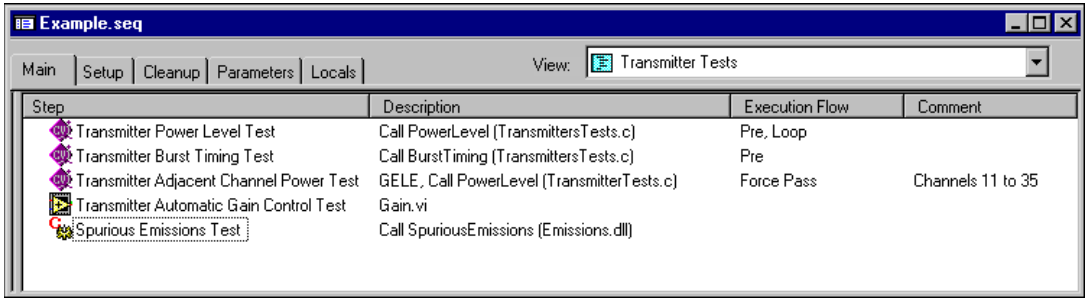

Figure 2-2. Example Sequence File Window

Refer to Chapter 5, *Sequence Files*, to learn more about editing a sequence file using the Sequence File window.

Execution Window

The sequence editor displays each execution in a separate window, called the Execution window. The Execution window is divided into several areas. The top half of the window contains tabs that display the Steps View, the Context View, and the Report View. The bottom half of the window is divided into the Call Stack View and the Watch Expression View. A status bar appears at the bottom edge of the window.

Figure 2-3 shows an example sequence editor Execution window.

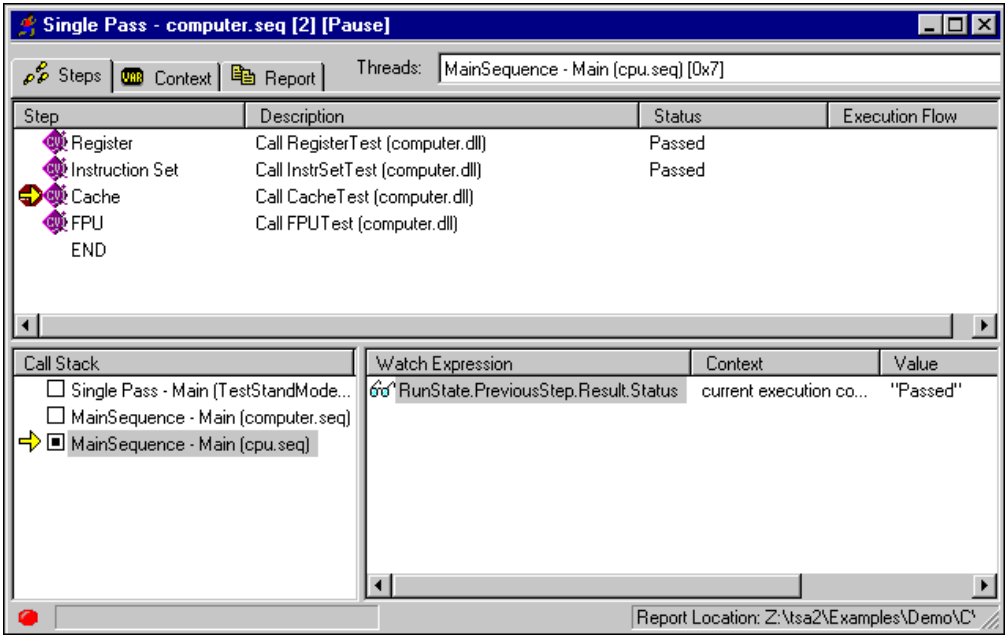

Figure 2-3. Example Execution Window

Refer to Chapter 6, *Sequence Execution*, to learn more about starting and debugging an execution using the Execution window.

Type Palette Window

You use the Type Palette window to store the data types and step types that you want to be available to you in the sequence editor at all times. The Type Palette window contains tabs for step types, custom data types, and standard data types.

Figure 2-4 shows an example Type Palette window.

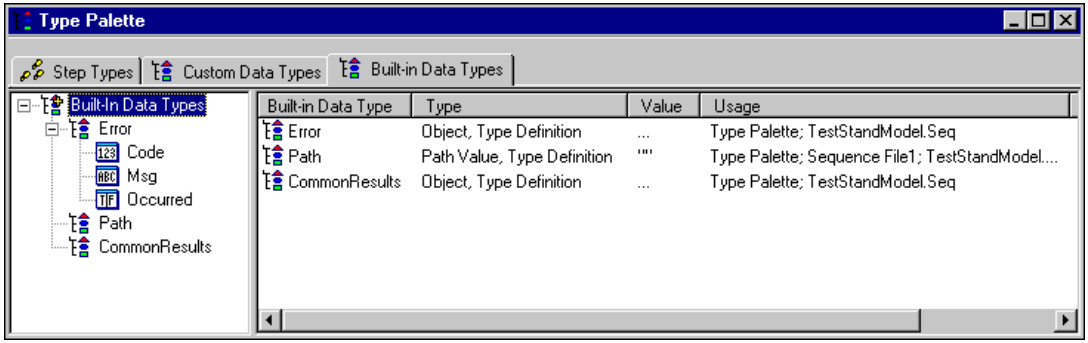

Figure 2-4. Example Type Palette Window

Refer to Chapter 9, *Types*, to learn more about using data types and step types.

Station Globals Window

The Station Globals window displays the variables that TestStand maintains from one TestStand session to the next. Usually, you use station global variables to maintain statistics or to represent the configuration of your test station.

Figure 2-5 shows an example Station Globals window.

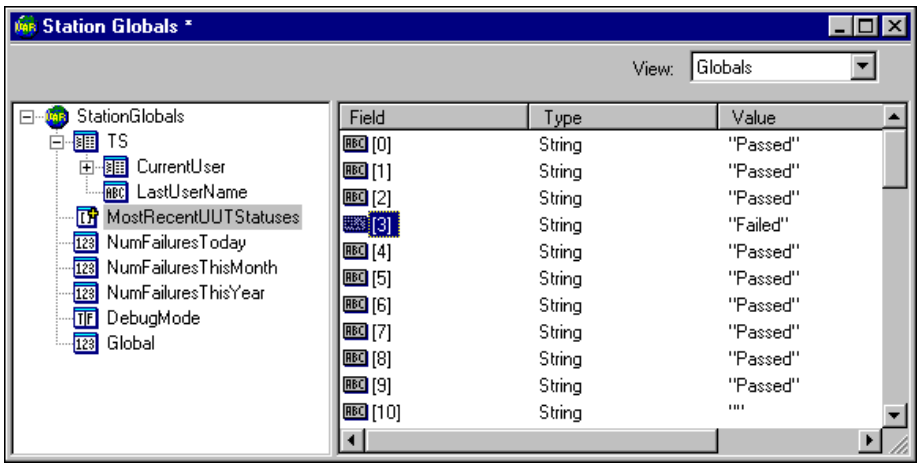

Figure 2-5. Example Station Globals Window

Refer to Chapter 7, *Station Global Variables*, to learn more about using station globals. Refer to Chapter 9, *Types*, to learn more about using data types.

Users Window

You can use the Users window to view and change the user list, user privileges, and the profiles for adding new users. Figure 2-6 shows an example Users window.

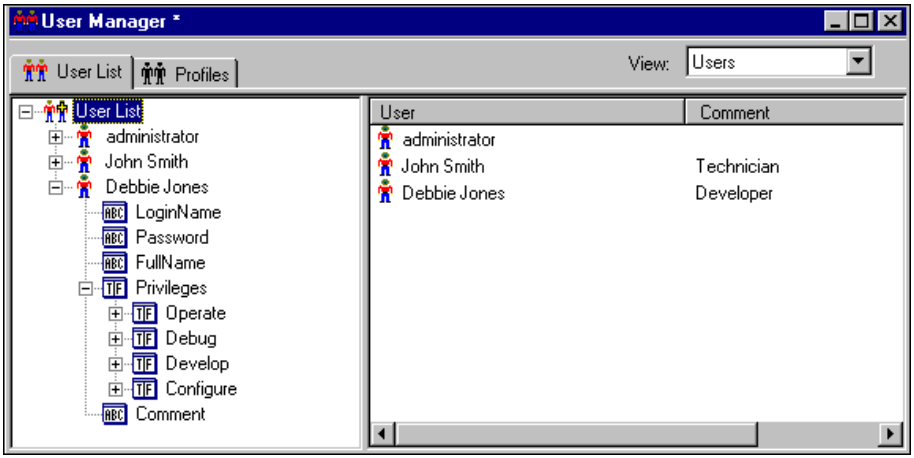

Figure 2-6. Example Users Window

Refer to the *User Manager Window* section in Chapter 11, *User Management*, to learn more about adding users and changing user privileges.

Basics of Using TestStand

This section describes how you perform some basic tasks in TestStand.

Creating a Sequence

In the sequence editor, you use a Sequence File window to view and edit a sequence file. You can open an existing sequence file by selecting **File»Open**, or you can create a new Sequence File window by selecting **File»New**.

You can use the View ring control at the top right of the Sequence File window to view an individual sequence, a list of all sequences in the file, the global variables in the file, or the types that you use in the file. In an individual sequence view, you use the tabs to view the Main, Setup, or Cleanup step groups, the sequence parameters, or the sequence local variables.

Figure 2-7 shows the Main step group of an example sequence in the Sequence File window.

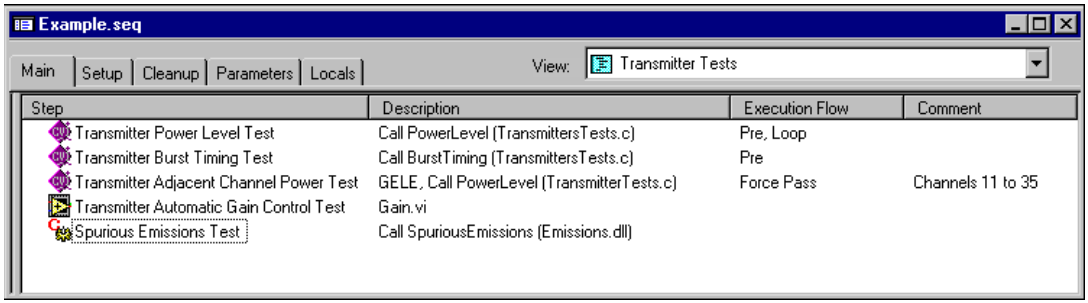

Figure 2-7. Main Step Group in an Example Sequence

Each step in a sequence has a step type. The step type defines the custom step properties and standard behavior for each step of that type. You can insert steps in the Main, Setup, and Cleanup tabs of an individual sequence view. The **Insert Step** item in the context menu displays a submenu of all the step types, including the step types that come with TestStand and any custom step types that you create.

Figure 2-8 shows the submenu for the **Insert Step** item.

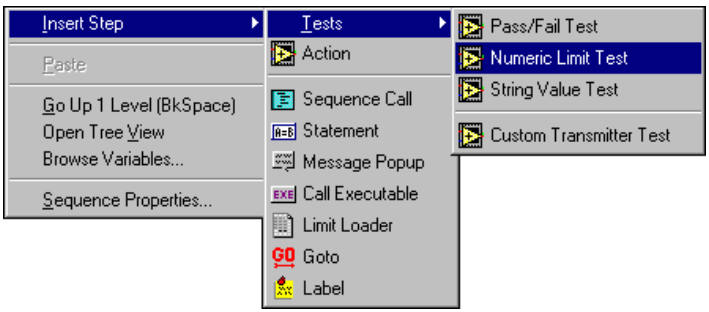

Figure 2-8. Insert Step Submenu

An icon appears to the left of each step type in the submenu. When you select a step type, TestStand displays the same icon next to the name of the new step in the list view. Many step types, such as the Pass/Fail Test and Action step types, can work with any module adapter. For these step types, the icon that appears in the submenu is the same as the icon for the module adapter that you select in the ring control on the tool bar. In Figure 2-8, the LabVIEW Standard Prototype Adapter is the current adapter, and its icon appears next to several step types, including Pass/Fail Test and Action. If you select one of these step types, TestStand uses the LabVIEW Standard Prototype Adapter for the new step.

Some step types require a particular module adapter and always use the icon for that adapter. For example, the Sequence Call step type always uses the Sequence Adapter icon. Other step types, such as Statement and Goto, do not use module adapters and have their own icons.

When you select an entry in the submenu, TestStand creates a step using the step type and module adapter that the submenu entry indicates. After you insert the step, you use the **Specify Module** item in the context menu for the step to specify the code module or sequence, if any, that the step calls. The **Specify Module** command displays a dialog box that is different for each adapter. Some adapters require you to specify the values to pass as arguments when executing the code module. Refer to Chapter 12, *Module Adapters*, for information on the Specify Module dialog box for each adapter.

For each step type, another item can appear in the context menu above **Specify Module**. For example, the **Edit Limits** item appears in the context menu for Numeric Limit Test steps, and the **Edit Destination** item appears in the context menu for Goto steps. You use this menu item to modify step properties that are specific to the step type. Refer to Chapter 10, *Built-In Step Types*, for information on the menu item for each step type.

To modify step properties that are common to all step types, use the **Properties** command in the context menu, double-click on the step, or press <Enter> when the step is selected.

Figure 2-9 shows the Step Properties dialog box.

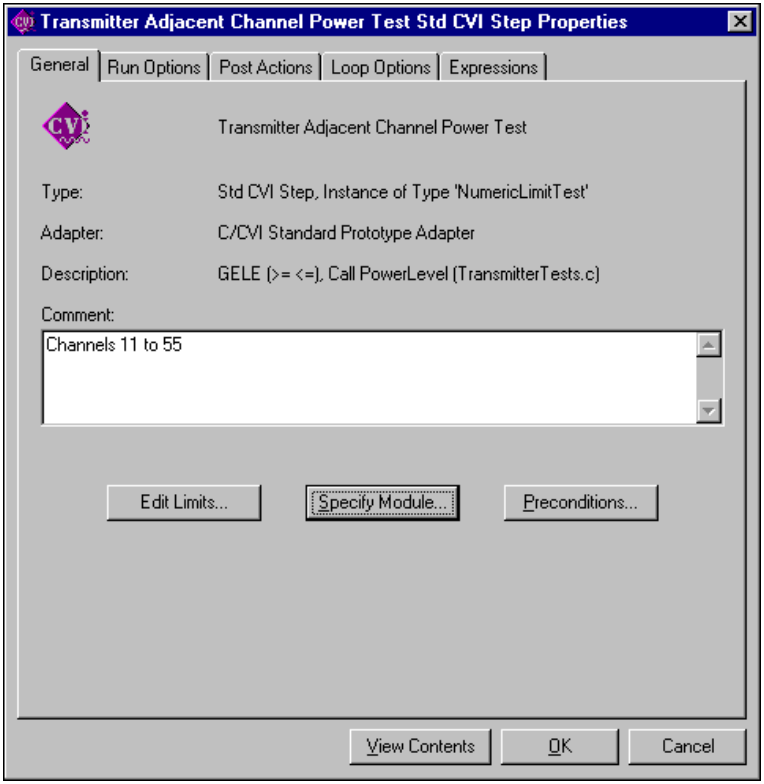

Figure 2-9. Step Properties Dialog Box

The Step Properties dialog box contains the following five tabs:

- **General**—Contains buttons to display the Specify Module dialog box, the step-type-specific dialog box and the Preconditions dialog box.
- **Run Options**—Specifies various options for loading and running the step code module.
- **Post Actions**—Specifies what action to take when the step finishes executing.
- **Loop Options**—Specifies whether the TestStand loops on the step. TestStand can loop a fixed number of times or loop until a specified number of iterations pass or fail. You also can customize the loop conditions.
- **Expressions**—Specifies expressions that TestStand executes before and after the step executes.

Refer to Chapter 5, *Sequence Files*, for more information on sequence files and adding steps to sequences.

Controlling Sequence Flow

TestStand has several features you can use to control the flow of execution in a sequence. These include the post actions for a step, the preconditions for a step, and the Goto step type. You can combine these features in various ways. For example, you can use the preconditions on a Goto step to specify when to loop back to an earlier statement.

Every step in TestStand has a status property. The status property is a string that indicates the result of the step execution. Although TestStand imposes no restrictions on the values to which the step or its code module can set the status property upon completion, TestStand recognizes the values that appear in Table 2-2.

Table 2-2. Standard Values for the Status Property after Execution Completes

Value	Meaning
`Passed`	Indicates that the step performed a test that passed.
`Failed`	Indicates that the step performed a test that failed.
`Error`	Indicates that a run-time error occurred.
`Done`	Indicates that the step completed without setting its status.
`Terminated`	Indicates that the step called a subsequence that terminated.
`Skipped`	Indicates that the step did not execute.

The post actions and preconditions you define can use the step status to control the flow of execution.

Post Action

You can use the Post Actions tab on the Step Properties dialog box to specify an action that occurs after the step executes. You can make the action conditional on the Pass/Fail status of the step or on any custom condition. Your choices of actions include:

- **Goto next step**—Execution continues normally with the next step. This is the default value.
- **Goto destination**—Execution branches to the destination you select. You can branch to any step in the current step group, to the end of the current step group, or to the Cleanup step group. If the post action for a step specifies that execution branches to the Cleanup step group and the current step is in the Cleanup step group, execution proceeds normally with the next step in the Cleanup group.
- **Terminate execution**—Execution terminates. Refer to the *Terminating and Aborting Executions* section in Chapter 1, *TestStand Architecture Overview*, for more information on execution termination.
- **Call sequence**—TestStand calls a sequence before continuing to the next step. You can select any sequence in the sequence file. TestStand does not pass any arguments to the sequence. If the sequence has parameters, TestStand uses their default values.
- **Break**—TestStand breakpoints before continuing to the next step.

Refer to the *Step Properties Dialog Box* section in Chapter 5, *Sequence Files*, for more information on the Post Action tab of the Step Properties dialog box.

Preconditions

The preconditions for a step specify the conditions that must be true for TestStand to execute the step during the normal flow of execution in a sequence. You can display the Preconditions dialog box by clicking on the **Preconditions** button on the Sequence Properties dialog box or by clicking on the **Preconditions** button on the Step Properties dialog box.

Figure 2-10 shows the Preconditions dialog box.

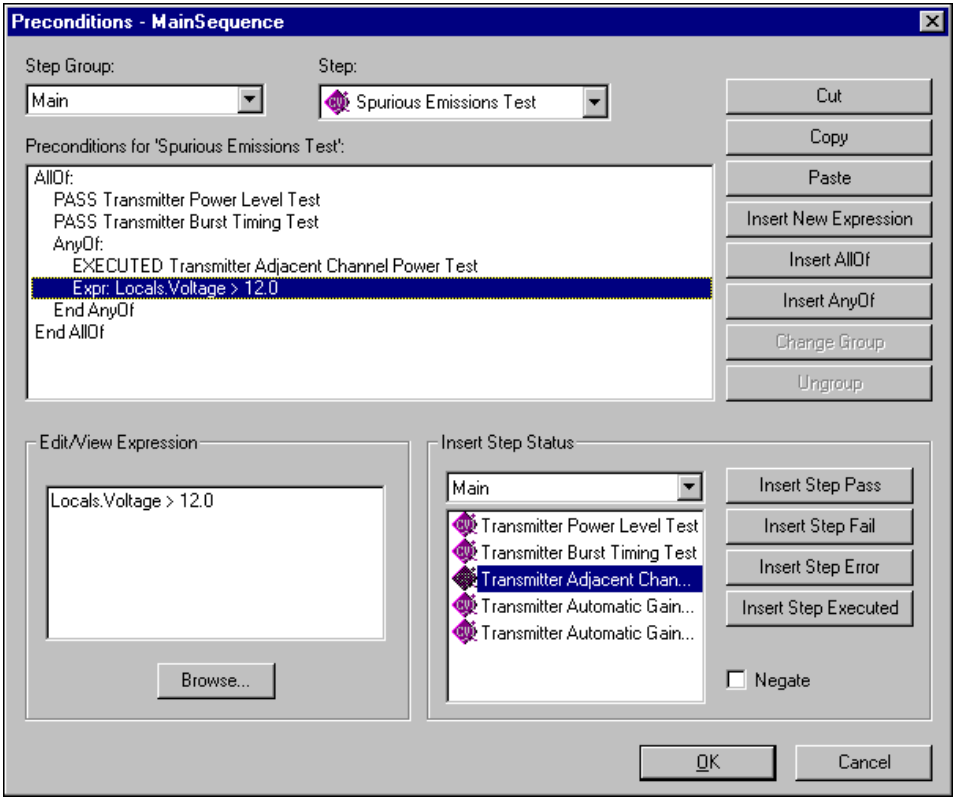

Figure 2-10. Preconditions Dialog Box

You can use a simple step status comparison as a condition. For example, you might want to execute a step only when the Power On Test passes. The Preconditions dialog box has special controls to make this easy. You also can specify an arbitrary expression that TestStand evaluates at run time. For example, you might want to execute the Keyboard test only when the `Locals.KeyboardInstalled` variable is `True`. You also can create complex preconditions by grouping conditions with the `All Of` and the `Any Of` operators. The `All Of` operator evaluates to `True` when all conditions in its group are `True`. The `Any Of` operator evaluates to `True` when at least one condition in its group is `True`.

Refer to the section *Preconditions Dialog Box* in Chapter 5, *Sequence Files*, for more information on how to use preconditions.

Goto Built-In Step Type

You use Goto steps to set the next step that the TestStand engine executes. You usually use a Label step as the target of a Goto step. This allows you to rearrange or delete steps in a sequence without having to change the target names in Goto steps. You can specify the Goto step target by selecting the **Edit Destination** item from the step context menu or the **Edit Destination** button on the Step Properties dialog box.

Refer to the *Goto* section in Chapter 10, *Built-In Step Types*, for more information on how to use the Goto step type.

Run-Time Errors

When a run-time error occurs in a step, execution in the sequence jumps to the Cleanup step group. After the Cleanup step group completes executing, TestStand reports the run-time error to the sequence call step in the calling sequence. This process continues up through the top-level sequence. Thus, when a run-time error occurs, TestStand terminates execution after running all the Cleanup steps of the sequences that are active at the time of the run-time error.

Running a Sequence

You can initiate an execution by launching a sequence through a model entry point, by launching a sequence directly, or by executing a group of steps interactively.

A list of entry points appears in the **Execute** menu of the sequence editor and operator interfaces. Each 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 entry point from the **Execute** menu, you actually run an entry point sequence in a process model file. The entry point sequence, in turn, invokes the main sequence in the active sequence file. The default TestStand process model provides two execution entry points: `Test UUTs` and `Single Pass`. The `Test UUTs` entry point initiates a loop that repeatedly identifies and tests UUTs. The `Single Pass` entry point tests a single UUT without identifying it.

To execute a sequence without using a process model, select the **Run *Sequence Name*** item in the **Execute** menu, where ***Sequence Name*** is the name of the sequence you are currently viewing. This command executes the sequence directly, skipping the process model operations such as UUT identification and test report generation. You can execute any sequence this

way, not just main sequences. Usually, you execute a sequence in this way to perform unit testing or debugging.

You can execute selected steps in a sequence interactively by choosing **Run Selected Steps** or **Loop Selected Steps** from the context menu in the sequence editor or by clicking on the **Run Tests** or **Loop Tests** buttons in the run-time operator interfaces. When you run steps interactively, TestStand does not evaluate step preconditions. If you execute steps in a Sequence File window, you initiate the interactive execution as an independent, top-level execution. If you execute steps in an Execution window when the execution is suspended, you initiate a nested interactive execution.

When you start a new execution, the sequence editor creates a new Execution window. Run-time operator interface programs update a view or create a new window for each new execution.

Refer to Chapter 13, *Process Models*, for more information on process models. Refer to Chapter 6, *Sequence Execution*, for more information on starting executions.

Debugging a Sequence

TestStand has several features you can use to debug the execution in a sequence. These include tracing, breakpoints, single-stepping, the sequence context browser, and watch expressions.

If tracing is enabled, the sequence editor and operator interfaces display the progress of an execution by highlighting the currently executing step in a step view. Usually, you disable tracing if you want to avoid using computer time to display the progress of your execution. You can use the **Tracing Enabled** item in the **Execute** menu to enable or disable tracing. You can set tracing options on the Execution tab in the Station Options dialog box. Refer to the *Configure Menu* and *Execute Menu* sections in Chapter 4, *Sequence Editor Menu Bar*, for more information on the tracing options.

The sequence editor and operator interfaces allow you to set breakpoints, to step into or step over steps, to step out of sequences, and to set the next step to execute. You also can terminate execution, abort execution, and run or loop on selected steps while at a breakpoint. In the sequence editor, these commands are in the **Debug** menu. Refer to the *Debug Menu* section in Chapter 4, *Sequence Editor Menu Bar*, for more information on debugging commands.

When using the sequence editor, you can display the variables and properties during an execution by selecting the Context view of the Execution window. The Context view displays the sequence context for the sequence invocation that is currently selected in the call stack. The sequence context contains all the variables and properties that the steps in the selected sequence invocation can access. You use the Context view to examine and modify the values of these variables and properties.

You can drag individual variables or properties from the Context view to the *Watch Expression* view so that you can view changes in specific values while you single-step or trace through the sequence.

Refer to Chapter 8, *Sequence Context and Expressions*, for more information on sequence contexts. Refer to Chapter 6, *Sequence Execution*, for more information on running and debugging an execution.

Generating Test Reports

TestStand automatically collects the results of an execution. As each step executes, TestStand appends the results from the step to a tree of results for an entire execution. When an execution completes, the default process model can generate a report from the information stored in the result tree. By default, an execution generates a report only when you start the execution through a model entry point such as `Test UUTs` or `Single Pass`.

You can set options that control report generation by selecting the **Report Options** item in the **Configure** menu. You can select either *HTML* or *ASCII* text formats. You can specify the report file name and whether TestStand generates the file name from the sequence file name, the time and date, or the UUT serial number. You can specify a result-filtering expression. For example, you can choose to include results only for steps that fail during an execution. You can specify whether TestStand appends results to the file if it already exists. You also can specify whether the report includes output values, test limits and execution times.

The Report view of the sequence editor Execution window displays the report for the current execution. Usually, the Report view is empty until execution completes. You also can use an external application to view reports by selecting the **Launch External Viewer** command from the **View** menu. You can use the **External Viewers** menu item in the **Configure** menu to specify the external application that TestStand launches to display a particular report format.

Figure 2-11 shows an HTML report for an example sequence.

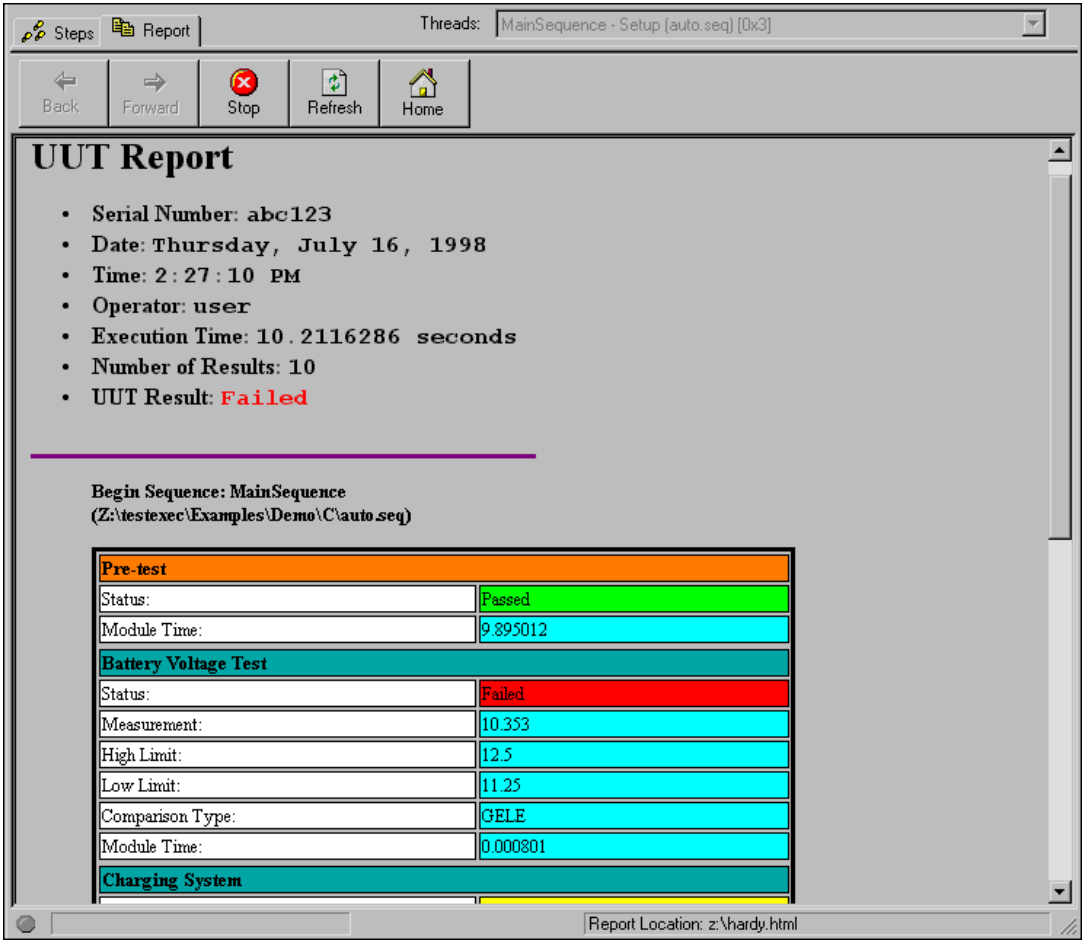

Figure 2-11. HTML Report for an Example Sequence.

Refer to the *Result Collection* section in Chapter 6, *Sequence Execution*, for more information on how TestStand collects results. Refer to Chapter 14, *Managing Reports*, for more information on available report options and customizing the report output.

Using an Operator Interface

Although you can use the TestStand sequence editor at a production station, the TestStand run-time operator interfaces are simpler. Also, they come with full source code so that you can customize them. Like the sequence editor, the run-time operator interfaces allow you to start multiple, concurrent executions, set breakpoints, and single-step. Refer to Chapter 15, *Run-Time Operator Interfaces*, in this document for more information on the operator interfaces included with TestStand.

3

Configuring and Customizing TestStand

This chapter summarizes how you can configure and customize a TestStand station.

Configuring TestStand

This section outlines the various configuration options in TestStand.

Sequence Editor Startup Options

You can append certain options to the sequence editor command line, separating various parameters by spaces. The valid startup options for the sequence editor appear in Table 3-1.

Table 3-1. Sequence Editor Startup Options

Option	Purpose
filename1 {*filename2*}…	The sequence editor automatically loads the sequence files at startup. Example: `SeqEdit "C:\MySeqs\seq1.seq" "C:\MySeqs\seq2.seq"`

Configure Menu

The **Configure** menu in the sequence editor and in the operator interfaces contains various commands to control the operation of the TestStand station. This section gives a brief overview of the item in the **Configure** menu. Refer the to *Configure Menu* section in Chapter 4, *Sequence Editor Menu Bar*, for more information on each menu item.

You can use the **Station Options** command to set preferences for your TestStand station. The settings affect all sequence editor and operator interface sessions that you run on your computer. The command displays a dialog box with the following tabs.

- **Execution**—Contains options for breakpoints, tracing, and interactive execution.
- **Time Limits**—Allows you to specify time limits for executions. If you specify a time limit, you choose an action to take when a time limit expires.
- **Preferences**—Specifies general options for the TestStand station, such as whether to save files before starting an execution.
- **Model**—Specifies the process model file for the station as a whole and whether each individual sequence can specify its own process model file.
- **User Manager**—Specifies whether TestStand enforces user privileges. It also specifies the location of the user manager configuration file.
- **Language**—Specifies language in which to show text.

The **Search Directories** command lets you customize the search paths for finding files. The dialog box displays a list of paths. The paths that appear first in the list take precedence over the paths that appear later. When you first run TestStand, the list contains a default set of directory paths.

The **External Viewers** command displays a dialog box in which you can specify the external viewer to use for each particular report format.

The **Adapters** command displays a dialog box in which you can configure a specific module adapter or specify the active module adapter. Refer to the *Configuring Adapters* section in Chapter 12, *Module Adapters*, for more information.

The **Report Options** command displays a dialog box in which you can customize the generation of report files. Refer to Chapter 13, *Process Models*, for more information on available report options.

Customizing TestStand

This section outlines the various methods you can use to customize a TestStand station.

TestStand Directory Structure

The TestStand installation program installs the TestStand engine, the sequence editor, the module adapters, and additional components on your system.

Table 3-2 shows the names and contents of each subdirectory.

Table 3-2. TestStand Subdirectories

Directory Name	Contents
`AdapterSupport`	Support files for the LabVIEW and C/CVI Standard Prototype Adapters.
`Api`	TestStand ActiveX Automation Server libraries for LabWindows/CVI and MFC.
`Bin`	TestStand sequence editor executable, engine DLLs, and support files.
`Cfg`	Configuration files for TestStand engine and sequence editor options.
`CodeTemplates`	Source code templates for step types. This directory contains an `NI` and a `User` subdirectory.
`Components`	Components that come with TestStand and components that you develop—This includes callback files, converters, icons, language files, process model files, step type support files, and utility files. This directory contains an `NI` and a `User` subdirectory.
`Doc`	Documentation files.
`Examples`	Example sequences and tests.
`OperatorInterfaces`	LabVIEW, LabWindows/CVI, and Visual Basic operator interfaces with source code. This directory contains an `NI` and a `User` subdirectory.
`Setup`	TestStand Installer/Uninstaller.
`Tutorial`	Sequences and code modules that you use in the tutorial sessions in the *Getting Started with TestStand* manual.

NI and User Subdirectories

Three of the TestStand directories contain source files that you might want to modify or replace. They are the `OperatorInterfaces`, `CodeTemplates`, and `Components` directories. Each directory contains an `NI` and a `User` subdirectory.

TestStand installs its files into the `NI` subdirectory. If you modify these files directly, the installers for newer versions of TestStand might overwrite your customizations. Consequently, it is best to keep the files you create or modify separate from the files that TestStand installs.

For this purpose, the TestStand installer creates a `User` subdirectory tree for you. Not only do you use the `User` subdirectory to protect your customized components, you use it as the staging area for the components that you include in your own run-time distribution of TestStand.

The Components Directory

TestStand installs the sequences, executables, project files, and source files for TestStand components in the `TestStand\Components\NI` directory. Most of the subdirectories under the `TestStand\Components\NI` directory have the name of a type of TestStand component. For example, the `TestStand\Components\NI\StepTypes` subdirectory contains support files for the TestStand built-in step types.

In general, if you want to create a new component or customize a TestStand component, copy the component files from the `NI` subdirectory to the `User` subdirectory before customizing. This ensures that the installers for newer versions of TestStand do not overwrite your customizations. If you copy the component files as the basis for creating a new component, make sure that you rename the files so that your customizations do not conflict with the default TestStand components.

The TestStand engine searches for sequences and code modules using the TestStand search directory path. The default search precedence places the `TestStand\Components\User` directory tree before the `TestStand\Components\NI` directory tree. This ensures that TestStand loads the sequences and code modules that you customize instead of loading the default TestStand versions of the files. You can modify the precedence of the TestStand search directory paths with the **Search Directories** command in the **Configure** menu of the sequence editor menu bar.

When you distribute a run-time version of the TestStand engine, you can bundle your components in the User directory with the TestStand run-time distribution. Refer to Chapter 16, *Distributing TestStand*, for more information on how to distribute the TestStand engine and your custom components.

Table 3-3 lists each subdirectory in the `NI` and `User` directory trees under `TestStand\Components`.

Table 3-3. TestStand Component Subdirectories

Directory Name	Contents
`Callbacks`	The `Callbacks` directory contains the sequence files in which TestStand stores station engine callbacks and front-end callbacks. TestStand installs the station engine and front-end callback files into the `TestStand\Components\NI\Callbacks` directory tree. Refer to *Customizing the Engine and Front-End Callbacks* section later in this chapter for more information on customizing the station and front-end callbacks.
`Icons`	The `Icons` directory contains icon files for module adapters and step types. TestStand installs the icon files for module adapters and built-in step types into the `TestStand\Components\NI\Icons` directory. Refer to the *Creating Step Types* section in this chapter for more information on creating your own icons for your custom step types.
`Language`	The `Language` directory contains string resource files. It has one subdirectory per language, for example, `English`. Refer to the *Creating String Resource Files* section in this chapter for more information on creating resource string files in the `Language` directory tree.
`Models`	The `Models` directory contains the default process model sequence files and supporting code modules. Refer to the *Modifying the Process Model* section in this chapter for more information on customizing the process model.
`RuntimeServers`	The `RuntimeServers` directory contains a LabVIEW run-time application for executing LabVIEW code modules. Refer to the *Customizing and Distributing a LabVIEW Run-Time Server* section in Chapter 16, *Distributing TestStand*, for more information on using LabVIEW run-time servers.

Table 3-3. TestStand Component Subdirectories (Continued)

Directory Name	Contents
`StepTypes`	The `StepTypes` directory contains support files for step types. TestStand installs the support files for the built-in step types into the `TestStand\Components\NI\StepTypes` directory tree. Refer to the *Creating Step Types* section in this chapter for more information on customizing your own step types.
`Tools`	The `Tools` directory contains sequences and supporting files for the **Tools** menu commands. Refer to the *Using the Tools Menu* section in this chapter for more information on customizing the **Tools** menu.

Creating String Resource Files

TestStand uses the `GetResourceString` function to obtain the string messages that it displays on windows and dialog boxes in the sequence editor and operator interfaces. `GetResourceString` works with string resource files that are in a `.ini` style format. `GetResourceString` takes a string category and a tag name as arguments. `GetResourceString` searches for the string resource in all string resource files that are in a predefined set of directories.

The directory search order is as follows:

1. `TestStand\Components\User\Language\<`*`current language`*`>`
2. `TestStand\Components\User\Language`
3. `TestStand\Components\NI\Language\<`*`current language`*`>`
4. `TestStand\Components\NI\Language\English`
5. `TestStand\Components\NI\Language`

You can change the current language setting by selecting **Configure»Station Options**.

TestStand installs the default resource string files in the `TestStand\Components\NI\Language` directory tree. If you want to customize a resource string file for a different language, you must copy an existing language file from the `NI` directory tree, place it in the `User` directory tree under a language subdirectory, and modify it. If you want to create a resource string file that applies to all languages, place the resource file in the base `TestStand\Components\User\Language` directory.

If you want to create your own resource string file for your custom components, make sure the category and tag names inside the resource file are unique so that they do not conflict with any that TestStand includes.

Resource String File Format

Each string resource file must have the `.ini` file extension. The format of a string resource file is as follows:

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

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

When you specify custom resource strings, you create the category and tag names. The number of categories and tags is unlimited.

A string can be of unlimited size. If a string has more than 512 characters, you must break it into multiple lines. Each line has a tag suffix of `line`*NNNN*, where *NNNN* is the line number with zero padding. The following is an example of a multiple-line string:

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

You can insert unprintable characters using escape codes. Table 3-4 lists the escape codes you can use.

Table 3-4. Resource String File Escape Codes

Escape Code	Description
`\n`	Linefeed character.
`\r`	Carriage return character.
`\t`	Tab character.
`\x`*nn*	Hexadecimal value. For example, `\x1B` represents the ASCII ESC character, which has a decimal value of 27.
`\\`	Backslash character.

For example, the following string contains an embedded linefeed character:

```
tag1 line0001 = "This is the first line.\nThis is the
   second line"
```

Using Data Types

You can use data types as station globals, sequence file globals, sequence locals, or properties of steps and step types. You can create and modify your own data types in TestStand. You also can modify the TestStand standard named data types by adding subproperties to them. Refer to the *Creating and Modifying Data Types* and *Using the Standard Named Data Types* sections in Chapter 9, *Types*, for more information.

Creating Step Types

If you want to change or enhance a TestStand built-in step type, do not edit the built-in step type or any of its supporting source code modules. Instead, copy and rename the built-in step type in the sequence editor. Also, copy its supporting modules from the `TestStand\Components\NI\StepTypes` directory tree to `TestStand\Components\User\StepTypes` directory. Make the changes to the copies. This ensures that you do not lose your changes when you install newer versions of TestStand. Refer to the *Using Step Types* section in Chapter 9, *Types*, for more information on step types and how you use them.

When creating a new step type, you can designate a specific icon to associate with that step type. The TestStand engine loads all available icons when you start the engine, so you must restart the sequence editor before you can associate a new icon with a step type. The `TestStand\Components\NI\Icons` directory contains icon files for the TestStand engine, the module adapters, and the built-in step types. If you want to override the `TestStand\Components\NI` icons or load icons for your custom step types, place the new icon file in the `TestStand\Components\User\Icons` directory and restart the engine. The TestStand engine loads all icons from the `TestStand\Components\User\Icons` and `TestStand\Components\NI\Icons` directories. If an icon of the same name is in both directories, the TestStand engine uses the one from the `TestStand\Components\User\Icons` directory. The TestStand engine does not search for icon files in any other directories.

Using the Tools Menu

The `TestStand\Components\NI\Tools` directory contains sequences and supporting files for the default TestStand **Tools** menu commands. The tools include a documentation generator, converters for LabVIEW and LabWindows/CVI Test Executive sequences, and compatibility tools for the LabVIEW and LabWindows/CVI Test Executive sequences that you convert.

If you want to create your own **Tools** menu commands, place any supporting code modules in the `TestStand\Components\User\Tools` directory tree. If you want to change or enhance a TestStand **Tools** menu command, do not edit the supporting source code modules. Instead, copy the files to `TestStand\Components\User\Tools` directory tree, and make the changes to this copy. This ensures that you do not lose your changes when you install newer versions of TestStand.

Refer to the *Tools Menu* section in Chapter 4, *Sequence Editor Menu Bar*, for more information on how to add your own commands to the **Tools** menu.

Refer to Chapter 16, *Distributing TestStand*, for more information on distributing a custom **Tools** menu with the run-time version of TestStand.

Customizing the Engine and Front-End Callbacks

The `TestStand\Components\NI\Callbacks` directory tree contains sequences and supporting files for the default TestStand front-end and station engine callbacks. TestStand installs the station engine callbacks in the `Station` subdirectory and the front-end callbacks in the `FrontEnd` subdirectory.

You can replace these callbacks individually. To do so, you must create a callback file in the `TestStand\Components\User\Callbacks` directory tree that has same name and relative location as the `NI` directory copy. For example, the `FrontEndCallbacks.seq` in the `TestStand\NI\Callbacks\FrontEnd` directory contains the default `LoginLogout` callback. You can override `LoginLogout` by creating a `LoginLogout` sequence in a new version of `FrontEndCallbacks.seq` in the `TestStand\User\Callbacks\FrontEnd` directory. TestStand then loads the `LoginLogout` sequence from the `User` directory instead of from the `NI` directory.

Refer to the *Callback Sequences* section in Chapter 1, *TestStand Architecture Overview*, for an overview of the different categories of callbacks. Refer to the *Engine Callbacks* section in Chapter 6, *Sequence Execution*, for more information on engine callbacks.

Note ***You must not define a*** `SequenceFileUnload` ***callback in the*** `FrontEndCallback.seq` ***or*** `StationCallbacks.seq` ***sequence files. When you do this, TestStand will hang when you shut down the TestStand engine.***

Modifying the Process Model

TestStand installs the default process model sequence file, `TestStandModel.seq`, and its supporting files into the `TestStand\Components\NI\Models\TestStandModel` directory.

If you want to change or enhance the default process model file, do not edit `TestStandModel.seq` or any of its supporting files. Instead, copy the files you want to change to a subdirectory that you name under the `TestStand\Components\User\Models` directory, and make the changes to the copies. This ensures that you do not lose your changes when you install newer versions of TestStand.

For example, if you want to change the HTML report output for all sequences, copy the `reportgen_html.seq` from the `NI` directory tree to the `User` directory tree and make changes to the new copy.

Refer to Chapter 13, *Process Models*, for more information on the default process model.

Refer to Chapter 14, *Managing Reports*, for more information on customizing the reports that TestStand generates.

Using Process Model Callbacks

Model callbacks allow you to customize the behavior of a process model for each main sequence that uses it. By defining one or more model callbacks in a process model, you specify which process model operations the sequence developer can customize. You can override the callback in the model sequence file by using the Sequence File Callbacks dialog box to create a sequence of the same name in the client sequence file.

Refer to the *Process Models* section in Chapter 1, *TestStand Architecture Overview*, for an overview of process models and model callbacks. Refer to Chapter 13, *Process Models*, for more information on the default process model and its callbacks.

Creating Code Templates

When creating step types, you can associate one or more code templates with the step type. Each code template has a name. TestStand installs its code templates under the `TestStand\CodeTemplates\NI` directory tree, where each subdirectory name is the name of a code template. Each code template has different source code files for each module adapter. TestStand stores the source files for the different module adapters in the template subdirectory. TestStand also maintains a `.ini` file in each template subdirectory. The `.ini` file contains a description string that TestStand uses for the code template. TestStand installs a default template in the `Default_Template` subdirectory.

Refer to the *Code Templates Tab* section in Chapter 9, *Types*, for more information on using and creating your own code templates.

Modifying Run-Time Operator Interfaces

TestStand installs the executable, project, and source files for each run-time operator interface in the `TestStand\OperatorInterfaces\NI` directory tree. If you want to customize one of the run-time operator interfaces, copy the operator interface project and source files from the `NI` subdirectory to the `TestStand\OperatorInterfaces\User` subdirectory before customizing. This ensures that you do not lose your customizations when you install newer versions of TestStand.

Refer to Chapter 15, *Run-Time Operator Interfaces*, for more information on the operator interfaces that ship with TestStand.

Adding Users and Managing User Privileges

You can add users to the TestStand user list by selecting **Configure»User Manager**. Refer to Chapter 11, *User Management*, for more information on adding new users, changing user privileges, and adding new user privileges.

4

Sequence Editor Menu Bar

This chapter describes the menu items in the sequence editor menu bar.

Menus

The sequence editor menu bar contains commands that apply to the entire test station. This section contains descriptions of the menu items in the sequence editor menu bar. For some commands, the description summarizes the features of the command and refers you to additional information later in this document.

File Menu

This section describes the **File** menu, as shown in Figure 4-1.

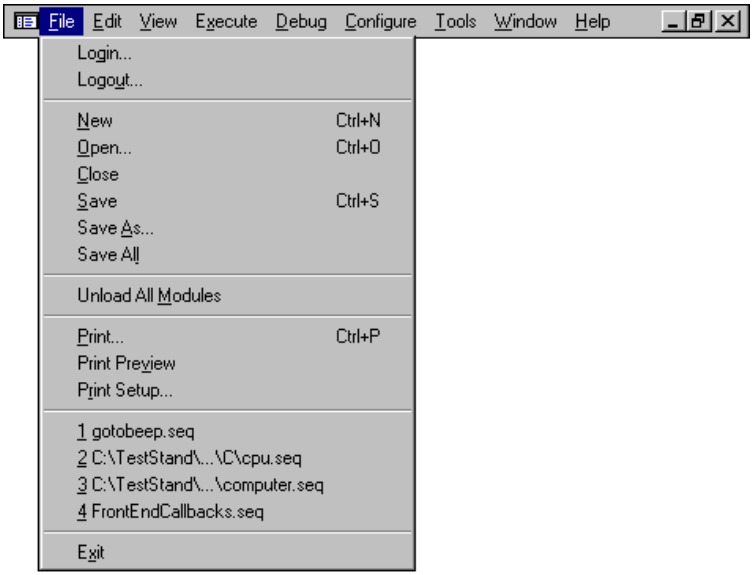

Figure 4-1. File Menu

Login

The **Login** command, which automatically executes when you open the sequence editor, prompts you for a login name and password. If you cancel out of the dialog box, you have no privileges. Use the **Login** command to log in as a different user. Each user can have different privilege settings, so logging in as a different user can change your privileges. Refer to Chapter 11, *User Management*, for more information.

Logout

The **Logout** command logs out the current user and displays the Login prompts.

New

Use the **New** command to create a new Sequence File window.

Open

Use the **Open** command to open an existing sequence file. When you select **Open**, a dialog box appears prompting you for a filename to load into a new window.

Close

Use the **Close** command to close an existing window. When you select **Close**, a dialog box might appear, prompting you to save any changes before closing the window.

Save

Use the **Save** command to write the contents of the active Sequence File window to disk.

Save As

Use the **Save As** command to write the contents of the active Sequence File window to disk using a new name you specify. The title bar on the Sequence File window displays the new name.

Save All

The **Save All** command saves all open files to disk, which includes sequence files, globals, type palette, users, and configuration information.

Unload All Modules

The **Unload All Modules** command removes from memory all step code modules, all code modules that substeps call, and all sequence files that are not currently in a window. You can use this command only if no executions are active. This command is useful when you want to rebuild a DLL after an execution, but the DLL is still loaded in TestStand. The ADE that you use to build the DLL cannot write out the new contents of the DLL until TestStand unloads it.

Most Recently Opened Files

For your reference, a list of the most recently opened files appears in the **File** menu.

Exit

Use the **Exit** command to close the current sequence editor session. If you have modified any open files since the last save, or if any windows contain unnamed files, the sequence editor prompts you to save them.

Edit Menu

You use the items in the **Edit** menu for editing sequences and steps.

Figure 4-2 shows the **Edit** menu.

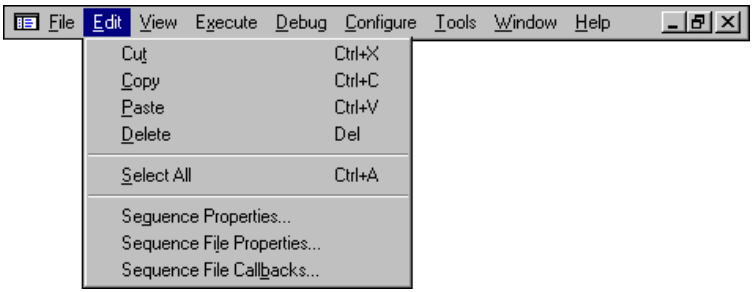

Figure 4-2. Edit Menu

Cut and Copy

The **Cut** and **Copy** commands place text or objects in the Clipboard. The **Cut** command removes the selected text or objects and places them in the Clipboard. The **Copy** command copies the selected text or objects and places them in the Clipboard, leaving them in their original location.

The text or objects that you cut or copy do not accumulate in the Clipboard. Every time you cut or copy text or objects, you replace the previous contents of the Clipboard.

Paste

The **Paste** command inserts text or objects from the Clipboard. You can paste text or objects from the Clipboard as many times as you like. The text or objects that you paste remain in the Clipboard until you use **Cut** or **Copy** again.

Delete

The **Delete** command deletes selected text or objects without replacing the contents of the Clipboard. Because **Delete** does not place the selected text or objects in the Clipboard, you cannot restore them using the **Paste** command.

Select All

The **Select All** command highlights all the objects in the active window.

Sequence Properties

The **Sequence Properties** command displays the properties for a selected sequence in the active Sequence File window, as shown in Figure 4-3. Refer to the *Sequence View Context Menu* section in Chapter 5, *Sequence Files*, for more information.

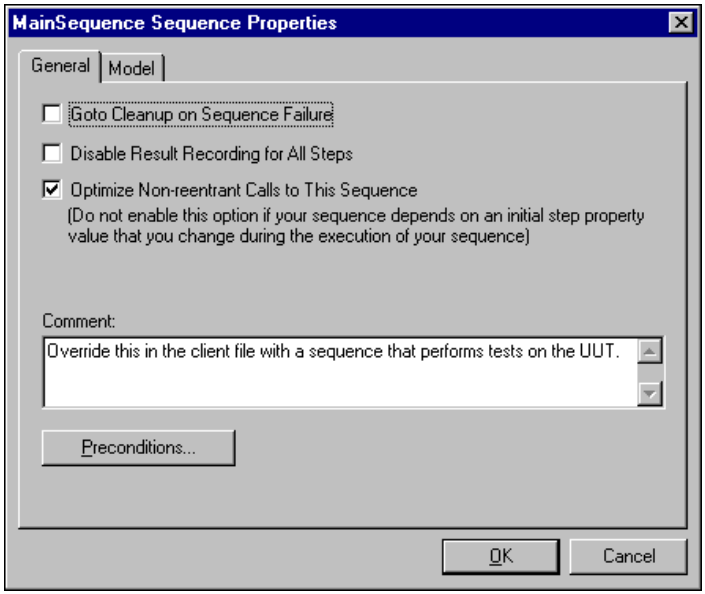

Figure 4-3. Sequence Properties Dialog Box

Sequence File Properties

The **Sequence File Properties** command displays the pathname, disk size, and disk date of the active sequence file. You also can use it to edit various properties of a sequence file, including the load and unload options, a comment, and the sequence file type. If the sequence file type is normal, you also can specify a process model file for the sequence file.

Figure 4-4 shows the Sequence File Properties dialog box.

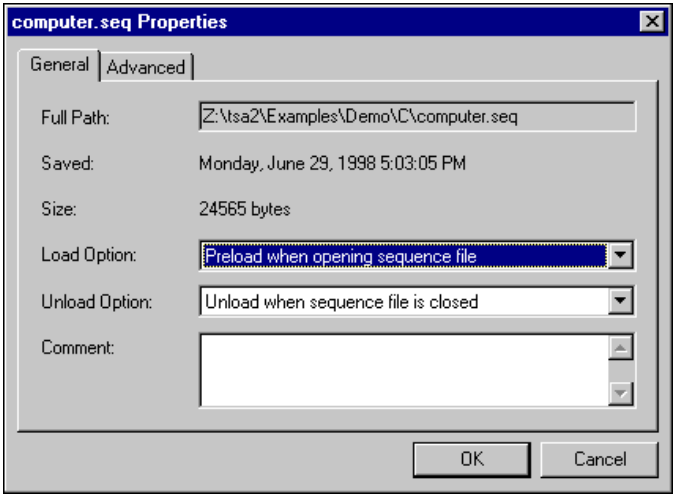

Figure 4-4. Sequence File Properties Dialog Box

Refer to the *Sequence View Context Menu* section in Chapter 5, *Sequence Files*, for more information on sequence file properties.

Sequence File Callbacks

The **Sequence File Callbacks** command displays a dialog box of all callbacks that you can override in the sequence file. This includes the Engine Callbacks that TestStand defines, and the Model Callbacks that the process model for the sequence file defines.

Figure 4-5 shows the Sequence File Callbacks dialog box.

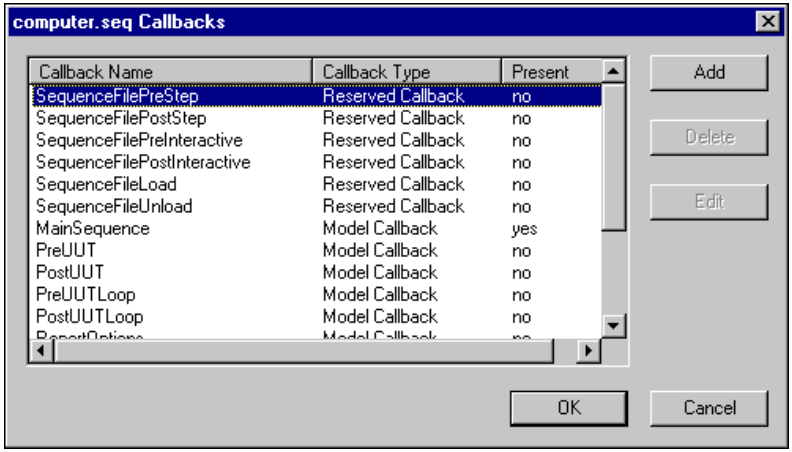

Figure 4-5. Sequence File Callbacks Dialog Box

Refer to the *Sequence View Context Menu* section in Chapter 5, *Sequence Files*, for more information on using the Sequence File Callbacks dialog box.

View Menu

This section explains how to use the commands in the **View** menu. Figure 4-6 shows the **View** menu.

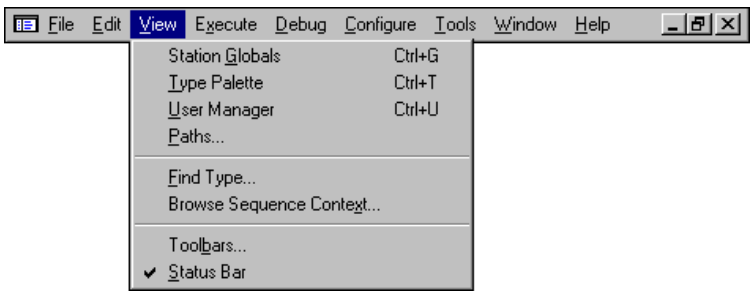

Figure 4-6. View Menu

Station Globals

The **Station Globals** command displays a window containing the station global variables and the types they use, including built-in and custom data types. For a detailed discussion of station globals, refer to Chapter 7, *Station Global Variables*.

Type Palette

The **Type Palette** command displays a window that contains a list of commonly used step types, built-in data types, and custom data types. For a detailed discussion of the types and the type palette, refer to Chapter 9, *Types*.

User Manager

The **User Manager** command displays a window for managing TestStand users and their privileges. From this window you can add new users, update user privileges, and change the types of privileges that users can have. For a detailed discussion of user management, refer to Chapter 11, *User Management*.

Paths

The **Paths** command lets you modify the directory portion of pathnames in sequence files and station configuration files. This can be useful after you copy a sequence file or configuration file from one computer to another.

Figure 4-7 shows the Edit Paths in Files dialog box.

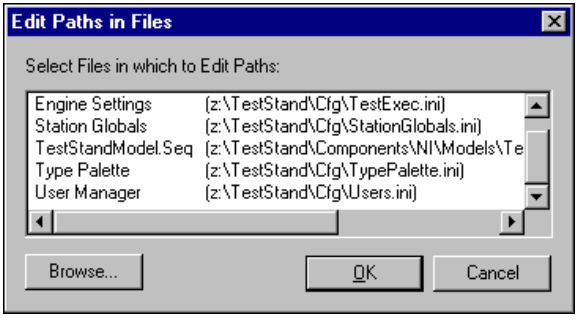

Figure 4-7. Edit Paths in Files Dialog Box

The list box contains three station configuration files and the sequence files currently in memory. For each of the sequence files in memory, the list box shows the simple filename and the complete pathname. For each of the station configuration files, the list box shows a symbolic tag and a pathname. The following are the symbolic tags and purposes of the station configuration files:

- `Config` is the file in which TestStand stores the station options.
- `Globals` is the file in which TestStand stores that names and values of the station global variables.
- `Type Palette` is the file in which TestStand stores the information in the type palette.

To edit the paths in one of the files in the list box, double-click on the file. To edit paths in multiple files in the list box, select the files and click on the **OK** button. To edit the paths in a sequence file that is not in the list box, use the **Browse** button.

When you select one or more files to edit, the Edit Paths dialog box appears. Figure 4-8 shows the Edit Paths dialog box.

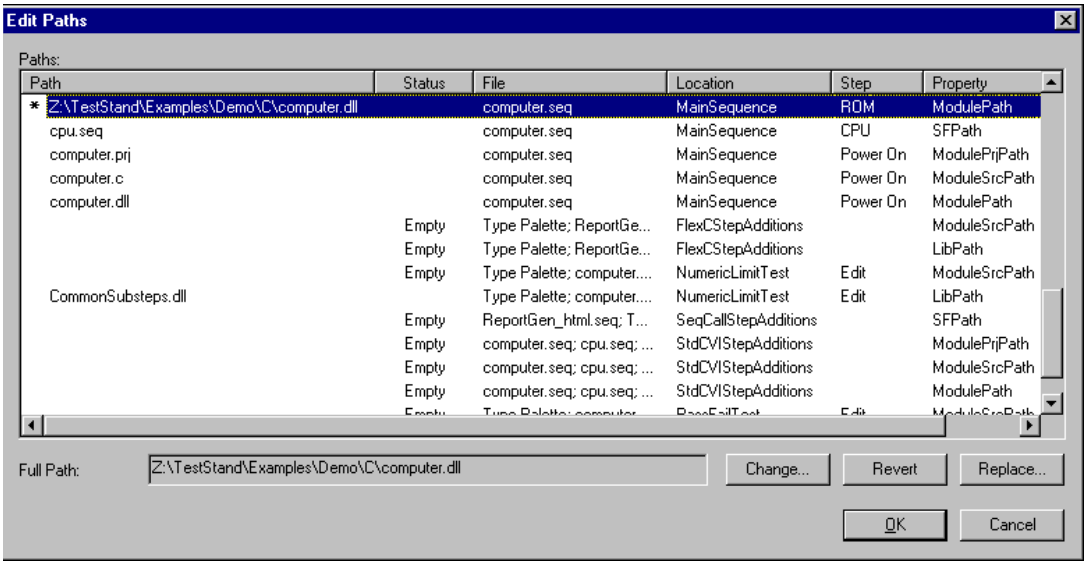

Figure 4-8. Edit Paths Dialog Box

Each entry in the list box represents a property that has the `Path` data type. The list box contains many columns of information for each entry. The Path column displays the current value of the property. The current value can be a simple pathname, a relative pathname, or an absolute pathname. The Status column displays `Empty` when the current value of the property is an empty pathname or when it is a pathname that TestStand cannot find on disk. The File column displays the name of the file that contains the property. The Sequence column displays the name of the sequence, if any, that contains the property. The location column specifies that particular part of the sequence that contains the property, for example, a step type or one of the step groups. The Step column displays the name of the step or substep, if any, that contains the property. The Property column displays the name of the property.

The Full Path indicator in the bottom left corner of the dialog box shows the absolute pathname to which TestStand can resolve the current value of the selected property. TestStand searches for the file through the search paths you specify using the Search Directories dialog box. If TestStand cannot find the file, the Full Path indicator shows `Not Found`.

Use the **Change** button to browse on disk for the file to which you want the property value to refer. If you find such a file, a dialog box prompts you to set the property value to the absolute pathname or to save the directory path in the list of search paths.

Use the **Revert** button to set the selected property to the value it had when you opened the dialog box.

Use the **Replace** button to change a substring in one or more of the property values. This is particularly useful for property values that are absolute pathnames. For example, if several properties contain pathnames that start with `c:\myfiles`, but the files are now in `d:\testfiles`, you can use the **Replace** button to change all instances of `c:\myfiles` in the list to `d:\testfiles`.

Find Type

The **Find Type** command displays a dialog box that contains a list of all types currently in memory. TestStand generates the list of types from all sequences currently in memory, the types that the user manager uses, the types that station global variables use, and types in the Type Palette window. You can jump to the window that contains a type by double-clicking on the type or by selecting the type and then clicking on the **Goto** button.

Figure 4-9 shows the Find Type dialog box.

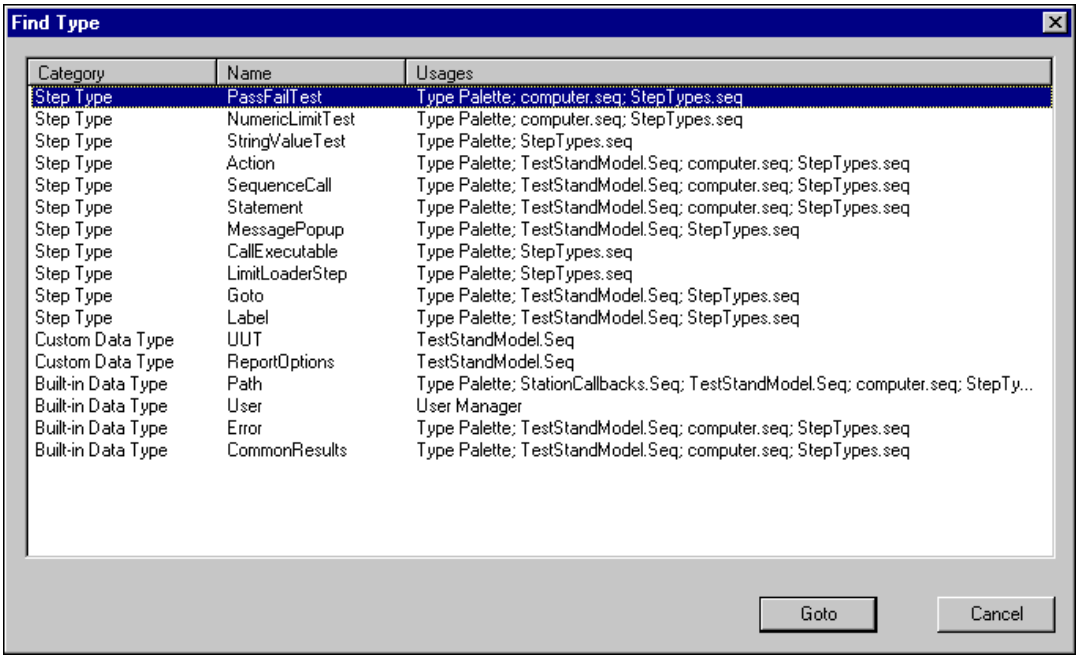

Figure 4-9. Find Type Dialog Box

Browse Sequence Context

The **Browse Sequence Context** command displays a tree view of variable and property names, as shown in Figure 4-10.

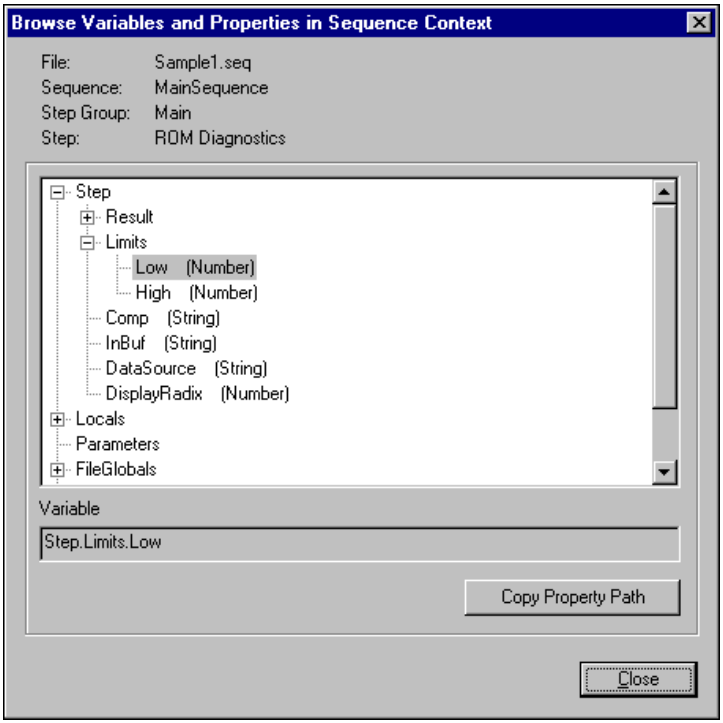

Figure 4-10. Browse Variables and Properties in Sequence Context Dialog Box

The tree view shows variables and properties that you can access in expressions or in step modules. The set of variable and property names that appears in the tree view depends on the active window and the currently selected item.

For example, the `Step` base property name appears when you select **Browse Sequence Context** on a step in the active sequence window. A text message above the tree view describes the active window and selected item.

Also, the variables and properties that you can access at run-time differ depending on the state of execution. A *sequence context* is the TestStand API object that contains the variables and properties you can access at a particular point during execution.

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

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

In step modules, you access the value of a variable or property by using `PropertyObject` methods in the TestStand ActiveX API on the sequence context. As with expressions, you must specify a path from the sequence context to the particular property or variable.

You can use the **Browse Sequence Context** command to build a string literal that specifies the path to a station variable, sequence file variable, sequence local variable, sequence parameter, or step property. Selecting **Copy Variable Name** copies the string literal to the system clipboard, which you can then paste into an expression or the code for a step module.

Toolbars

The **Toolbars** command displays a list of available toolbars. Visible toolbars have checkmarks beside them in the toolbar list.

Status Bar

Use the **Status Bar** command to specify whether the status bar is visible at the bottom of the main window. When the status bar is visible, a checkmark appears beside this command in the menu.

Launch Report Viewer

Use the **Launch Report Viewer** command to display the report for the current Execution window using the external viewer associated with the report format. This option is available only when an Execution window is active and the execution is complete.

Execute Menu

This section explains how to use the commands in the **Execute** menu, as shown in Figure 4-11.

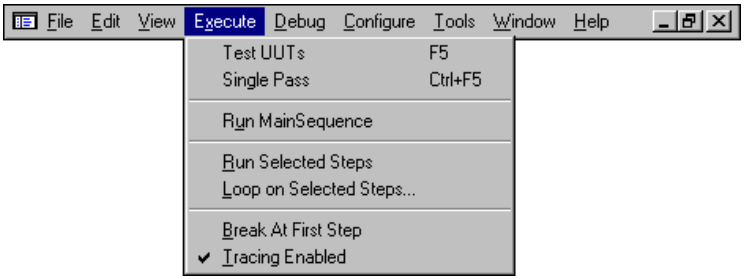

Figure 4-11. Execute Menu

Execution Entry Point List

Model execution entry points appear at the top of the **Execute** menu. For example, the default TestStand process model provides two entry points: `Test UUTs` and `Single Pass`. Selecting a model entry point invokes an execution using the active sequence. Refer to the *Starting an Execution* section in Chapter 6, *Sequence Execution*, for more information on using execution entry points to start an execution.

Run Active Sequence

Use the **Run Active Sequence** command to initiate an execution of the active sequence without using a process model.

Restart

Use the **Restart** command to rerun a completed execution. This option is available only when an Execution window is active and the execution is complete.

Run Selected Steps

You can execute selected steps in a sequence interactively by choosing **Run Selected Steps**. If you execute steps in a Sequence File window, you initiate the interactive execution as an independent, top-level execution. If you execute steps in an Execution window for a sequence execution that is suspended, you initiate the interactive execution as an extension of the suspended execution. When you run steps interactively, TestStand does not evaluate step preconditions.

Refer to the *Interactively Executing Steps* section in Chapter 6, *Sequence Execution*, for more information on running steps interactively.

Loop on Selected Steps

You can execute and loop on selected steps in a sequence interactively by choosing **Loop on Selected Steps**.

Figure 4-12 shows the Loop Count tab on the Loop on Selected Steps dialog box.

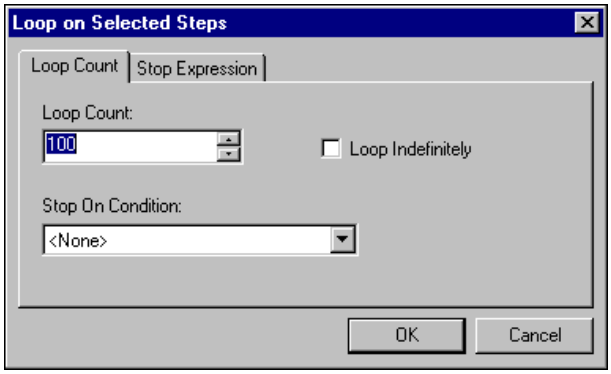

Figure 4-12. Loop on Selected Steps Dialog Box—Loop Count Tab

The Loop Count control specifies the maximum number of iterations that TestStand executes the selected steps. If you enable the Loop Indefinitely checkbox, the Loop Count control dims. You can also specify that TestStand stop the interactive execution when any step status is error, pass or fail. If you want TestStand to evaluate a custom expression after each step executes to determine whether TestStand continues the interactive execution, you can use the Stop Expression tab to specify the expression. The stop expression must evaluate to a Boolean value. TestStand stops looping if the stop expression evaluates to True.

Figure 4-13 shows the Stop Expression tab. When you enable the Specify Custom Stop Expression checkbox, the Stop On Condition control dims on the Loop Count tab.

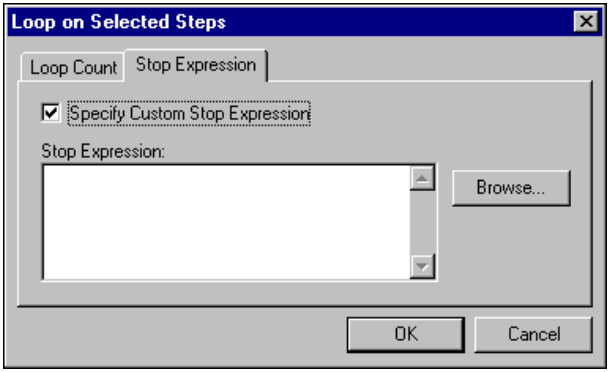

Figure 4-13. Loop on Selected Steps Dialog Box—Stop Expression Tab

If you execute steps in a Sequence File window, you initiate the interactive execution as an independent, top-level execution. If you execute steps in an Execution window for a sequence execution that is suspended, you initiate the interactive execution as an extension of the suspended execution. When you run steps interactively, TestStand does not evaluate step preconditions.

Refer to the *Interactively Executing Steps* section in Chapter 6, *Sequence Execution*, for more information on running steps in a loop interactively.

Break On First Step

Use **Break On First Step** to suspend execution on the first step that you execute whenever you initiate execution in the active sequence. When enabled, this command has a checkmark beside it in the menu.

Tracing Enabled

Use **Tracing Enabled** to highlight each step as it becomes the active step during execution. When disabled, updates to the sequence execution display occur only when execution is suspended. When enabled, this command has a checkmark beside it in the menu.

Debug Menu

This section explains how to use the commands in the **Debug** menu, as shown in Figure 4-14.

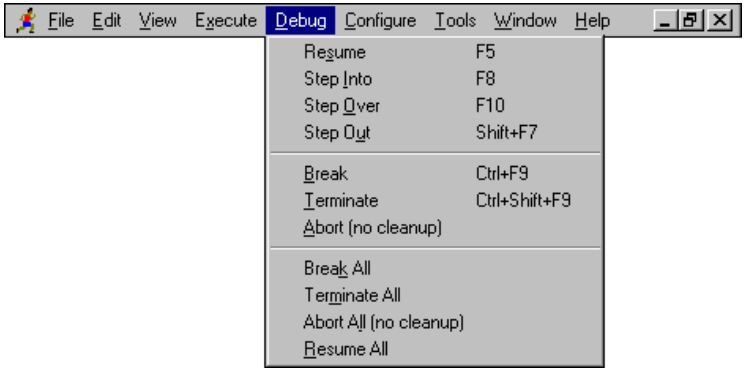

Figure 4-14. Debug Menu

Resume

Use the **Resume** command to continue execution when in a breakpoint state.

Step Over

Use the **Step Over** command to execute an outlined step when in a breakpoint state. If the program last suspended on a call to another sequence, **Step Over** executes the entire sequence and then enters a breakpoint state on the step following the sequence step. If the engine encounters a breakpoint within the sequence step, **Step Over** pauses at the breakpoint.

Step Into

The **Step Into** command is similar to the **Step Over** command except that **Step Into** enters the function and suspends at the first step in the sequence call.

Step Out

The **Step Out** command resumes execution through the end of the current sequence and breakpoints on the next step in the calling sequence.

Break

The **Break** command suspends the active execution after completing the execution of the current step.

Terminate

The **Terminate** command terminates a running or suspended execution. A running execution terminates only after completing the currently executing step. When you terminate an execution, TestStand runs the Cleanup step groups for all active sequences on the call stack.

Note ***If any of your step modules wait for user input or do not return quickly for any other reason, the step module can monitor for termination or abort requests by using the*** `Execution` ***class in the TestStand ActiveX API.***

Abort (no cleanup)

The **Abort** command aborts a running or suspended execution. A running execution aborts only after completing the currently executing step. When an execution aborts, TestStand does not run any Cleanup step groups.

Note ***If any of your step modules wait for user input or do not return quickly for any other reason, the step module can monitor for termination or abort requests by using the*** `Execution` ***class in the TestStand ActiveX API.***

Break All

The **Break All** command is similar to the **Break** command except that **Break All** suspends all running executions.

Terminate All

The **Terminate All** command is similar to the **Terminate** command except that **Terminate All** terminates all running executions.

Abort All (no cleanup)

The **Abort All** command is similar to the **Abort** command except that **Abort All** aborts all running executions.

Resume All

The **Resume All** command is similar to the **Resume** command except that **Resume All** continues all suspended executions.

Configure Menu

This section describes how to use the commands in the **Configure** menu, as shown in Figure 4-15.

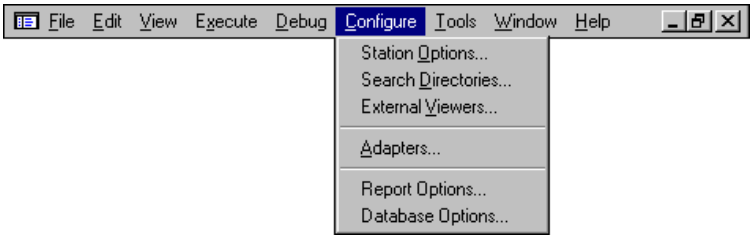

Figure 4-15. Configure Menu

Station Options

Use the **Station Options** command to set preferences for your TestStand station. The settings affect all sequence editor sessions you run on your computer. The command displays a dialog box with the following tabs.

Execution

The Execution tab has options for breakpoints, tracing, and interactive execution. Figure 4-16 shows the Execution tab.

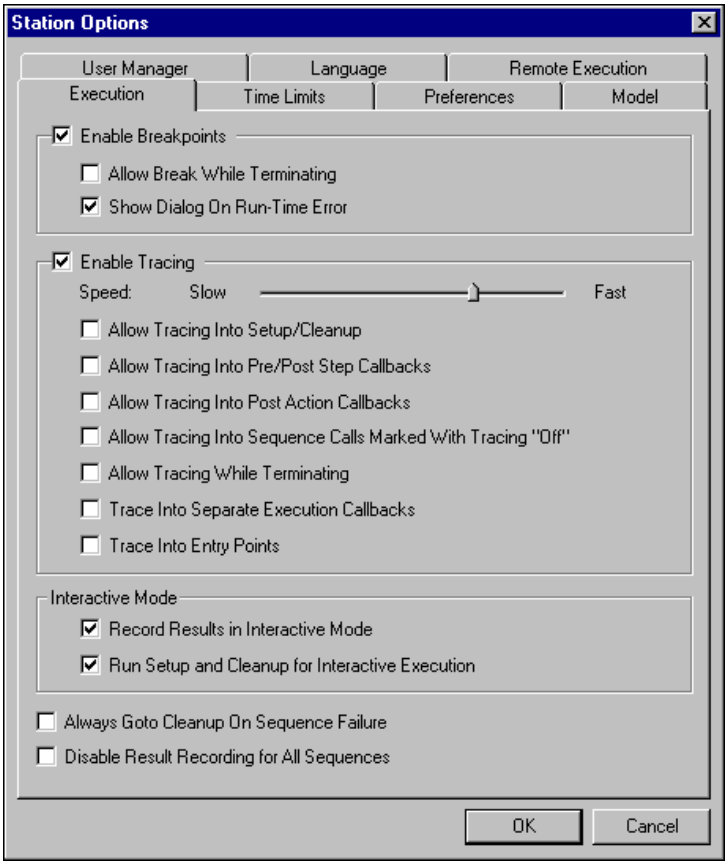

Figure 4-16. Execution Options

The following options are available on the Execution tab.

- **Enable Breakpoints**—You can use this option to enable or disable all breakpoints. When you enable breakpoints, the following additional options are available.
  - Allow Break While Terminating—Honors breakpoints when terminating an execution.
  - Show Dialog On Run-Time Error—Displays a dialog box when a run-time error occurs. The dialog box lists the failing step, the cause of the failure, and prompts you with options for handling the

error. The options include ignoring the failure and continuing execution, jumping to the Cleanup step group, and aborting immediately.

You also can choose to break at the current step and to suppress the run-time error dialog box during the current execution. Refer to the *Run-Time Errors* section in Chapter 6, *Sequence Execution*, for more information.

- **Enable Tracing**—You can use this option to enable or disable tracing. When tracing is in effect, the sequence editor or operator interface program displays each step as it is executing. This is useful for debugging but adds significant performance overhead to the execution of your test programs. When you enable tracing, the following additional options are available.
  - Speed—Specifies whether TestStand inserts a nominal delay between trace events sent to the sequence editor or any operator interface. This delay only applies when tracing is enabled. You can use this to slow down the tracing so that you can visibly see each step executing.
  - Allow Tracing Into Setup/Cleanup—Enables tracing of steps in the Setup and Cleanup step groups of each sequence.
  - Allow Tracing Into Pre/Post Step Callbacks—Enables tracing of steps in any of the Pre Step and Post Step Engine Callbacks.
  - Allow Tracing Into Post Action Callbacks—Enables tracing of steps in Post Action callbacks.
  - Allow Tracing Into Sequence Calls Marked With Tracing “Off”—Enables tracing into all subsequences when tracing is enabled for the calling sequence.

    In the Run Options tab of the Step Properties dialog box, you can choose a setting that disables tracing when the step calls a subsequence. If you enable the Allow Tracing Into Sequence Calls Marked With Tracing “Off” option in the Station options dialog box, TestStand ignores that Step Properties setting and does not alter the tracing state when it calls the subsequence.
  - Allow Tracing While Terminating—Enables tracing of steps that run while execution is terminating. Examples of steps that can run when execution is terminating are steps in Cleanup step groups that run when you terminate execution in the middle of a sequence.

  - Trace Into Separate Execution Callbacks—Enables tracing in callbacks that run as executions separate from the top-level sequence execution. Examples include front-end callbacks and callbacks that you execute from the **Tools** menu.
  - Trace Into Entry Points—Enables tracing of steps in process model point sequences, such as the `Test UUTs` and `Single Pass` entry points.
- **Interactive Mode**—Use this section to set options that apply when you run steps interactively
  - Record Results in Interactive Mode—If this option is enabled, TestStand records the results of steps that you run interactively. If you run steps interactively from an Execution window when suspended in a normal execution, TestStand appends the results to the result list for the active sequence invocation. Thus, the results appear in the test report for the normal execution.

    If you run steps interactively from a Sequence File window, TestStand accumulates the results in a result list for the interactive execution. Interactive executions do not use process models and thus do not generate test reports, but you can access an interactive execution result list from an Engine post-interactive callback.
  - Run Setup and Cleanup for Interactive Execution—Specifies whether to run the Setup and Cleanup step groups for the sequence that contains the selected steps. This option applies only when you run the steps from a Sequence File window.
- **Goto Cleanup On Sequence Failure**—Causes execution to jump to the Cleanup step group when the sequence status is set to Failure.
- **Disable Result Recording for All Sequences**—When you disable result recording with this option, the process model does not generate a result report for sequence executions.

Time Limits

The Time Limits tab allows you to specify time limits for executions. If you specify a time limit, you choose an action to take when a time limit expires. Figure 4-17 shows the Time Limits tab.

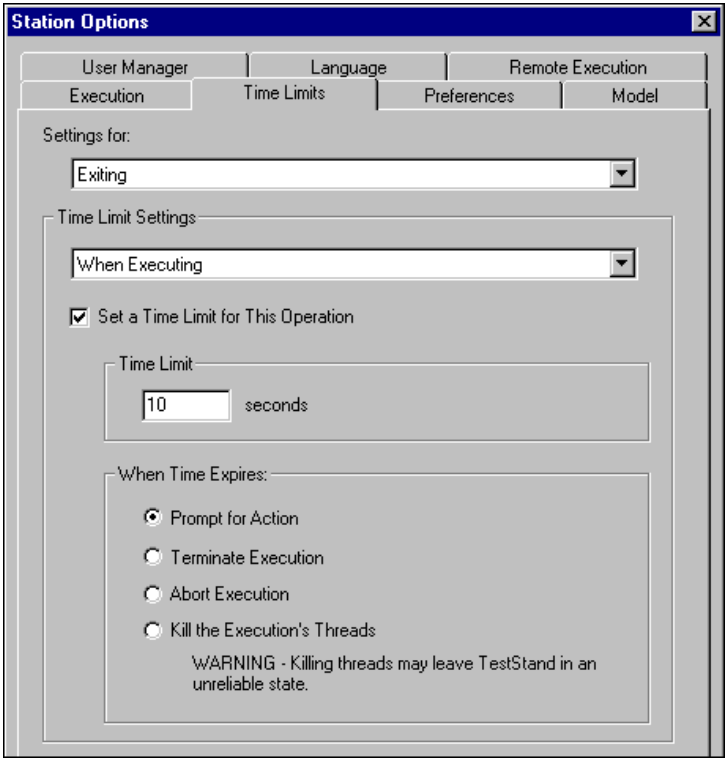

Figure 4-17. Time Limits Options

The tab maintains different time limits for normal execution and for executions that run while the engine is exiting. You can switch between the different time limits by using the Settings selection ring.

The Time Limit Settings selection ring contains the following types of time limits.

- **When Executing**—Applies to an execution from start to completion.
- **When Terminating**—Applies to executions from a termination request to completion.
- **When Aborting**—Applies to executions from an abort request to completion.

You can enable the time limit by enabling the Set a Time Limit for this Operation option.

You can select one of the following actions to take when the time limit expires.

- **Prompt for Action**—Displays a dialog box with the option to terminate, abort, or kill the execution.
- **Terminate Execution**—Initiates a termination of a running execution.
- **Abort Execution**—Initiates an abort of a running or terminating execution.
- **Kill the Execution's Threads**—Ends the thread for a running, terminating, or aborting execution.

When you terminate a running execution, TestStand executes all the Cleanup step groups in sequences on the call stack before execution stops. A terminating sequence can time out when a step in one of the Cleanup step groups hang or take a long time to complete. When you abort a running or terminating execution, TestStand returns back up the call stack without running any Cleanup step groups. An abort operation also can time out when the last executed step hangs or takes a long time to complete. When you *kill* a running, terminating, or aborting execution, TestStand terminates the thread running the execution without any cleanup of memory. This can leave TestStand in an unreliable state.

Preferences

The Preferences tab specifies general options for TestStand. Figure 4-18 shows the Preferences tab.

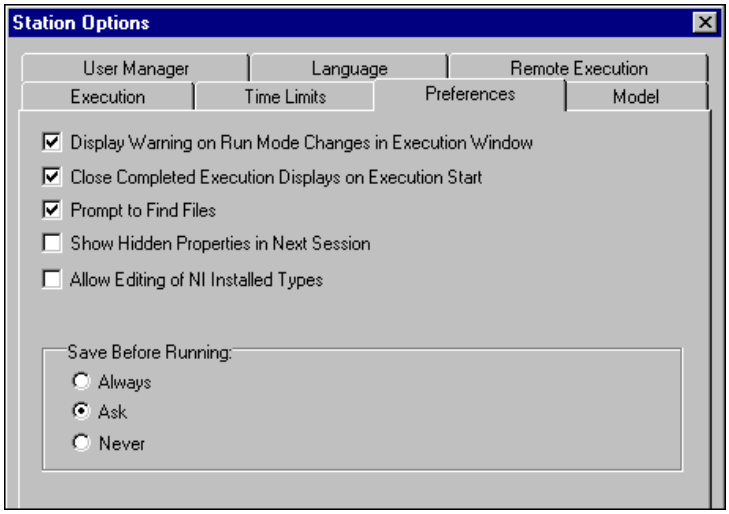

Figure 4-18. Preferences Options

The following options are available on the Preferences Tab.

- **Display Warning on Run Mode Changes in Execution Window**—Displays a warning dialog box when you modify the run mode for a step in an Execution window. When you modify the run mode in a Sequence File window, the modification applies to all subsequent executions, and TestStand writes the new run mode to disk when you save the sequence file. When you modify the run mode in an Execution window, the modification affects only that execution and TestStand does not save the modification to disk.
- **Close Completed Execution Displays on Execution Start**—Automatically closes all completed Execution windows when you start a new execution.
- **Prompt to Find Files**—Displays a file dialog box when TestStand cannot find necessary files in the current directory search path.
- **Show Hidden Properties in Next Session**—Enables the displaying of hidden properties. Most hidden properties are built-in step properties that TestStand uses. A change to this option does not take effect until the next time you restart the sequence editor.

- **Allow Editing NI Installed Types**—Allows users to modify the step and data types that ship with TestStand. These include built-in step types, standard data types, and some custom data types. If you attempt to edit an NI installed type, TestStand displays a dialog box stating that you must enabled this option to edit the type.
- **Save Before Running**—You can configure the sequence editor to never save modified files, to always save modified files, or to ask whether to save modified files before running.

Model

The Model tab specifies the model options for the station and for sequences. Figure 4-19 shows the Model tab.

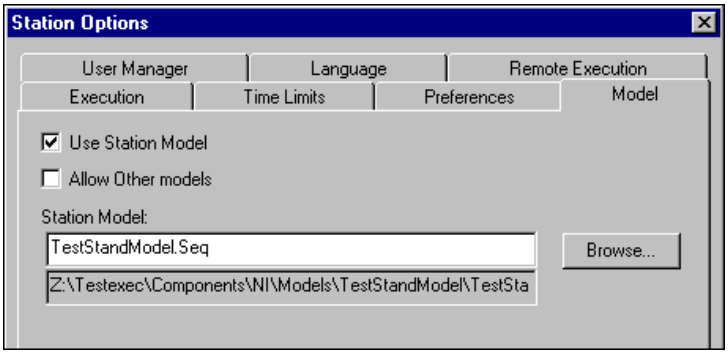

Figure 4-19. Model Options

The following options are available on the Model tab.

- **Use Station Model**—Enables the Station Model control, which specifies the pathname of the station model sequence file. When this option is disabled, no station model is in effect, and individual sequence files have no process model unless they specify one explicitly. Usually, sequence files do not specify process model files explicitly.
- **Allow Other Models**—Allows sequence files to specify a process model file other than the current station model file. When this option is disabled, you can load only sequence files that do not specify a process model file and sequences that specify the current station model file as their process model file.
- **Station Model**—Specifies the pathname of the station model sequence file.

User Manager

The User Manager tab specifies whether TestStand enforces user privileges. It also specifies the location of the user manager configuration file. Figure 4-20 shows the User Manager tab.

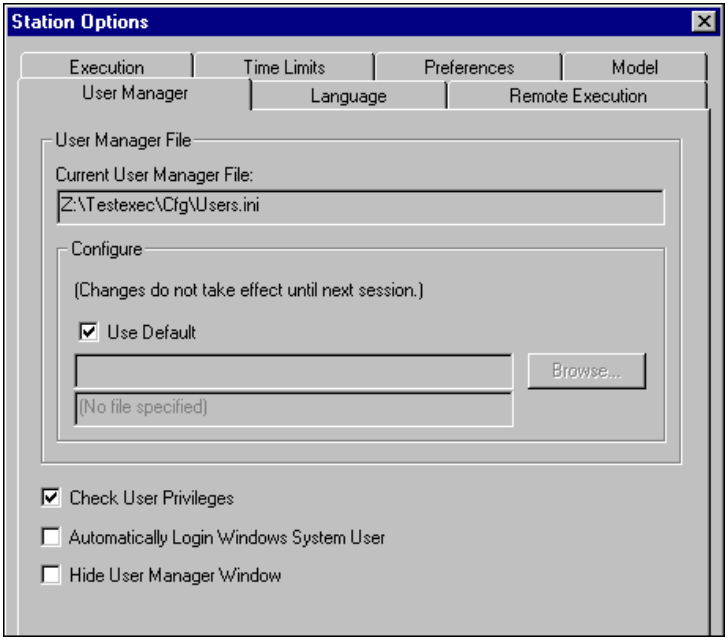

Figure 4-20. User Manager Options

The following options are available on the User Manager tab.

- **Check User Privileges**—Prevents users from accessing features for which they do not have privileges. If this option is disabled, any user can access any feature without regard to privileges. You must have sufficient privileges to change this option or any other option that affects privilege checking.
- **User Manager File**—Displays the user manager file that is currently in memory and allows you to select a new one. The default file is `TestStand\Cfg\Users.ini`. Selecting a new user manager file does not take effect until the next time you restart the sequence editor.
- **Hide User Manager Window**—Disables the **User Manager** menu and toolbar items.
- **Automatically Login Windows System User**—When enabled, TestStand attempts to login the current Windows user name in the TestStand user list. If the user name is found in the TestStand user list,

TestStand automatically logs in the user at the appropriate level without prompting for a password. If the user name is not found, TestStand prompts you to login. You must create a user entry in the TestStand User Manager and enter the Windows login name for the user as their TestStand login name for this option to function.

Note ***The TestStand User Manager is designed to help you implement policies and procedures concerning the use of your test station. It is not a security system and it does not inhibit or control the operating system or third-party applications. You must use the system-level security features provided by your operating system to secure your test station computer against malicious use.***

Language

The Language tab specifies the station language. Figure 4-21 shows the Language tab.

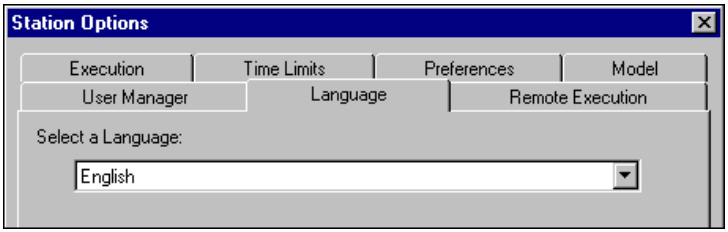

Figure 4-21. Language Options

Remote Execution

The Remote Execution tab specifies whether a remote machine can run a sequence on this station.

Search Directories

The **Search Directories** command lets you customize the search paths for finding files. Figure 4-22 shows the Edit Search Directories dialog box.

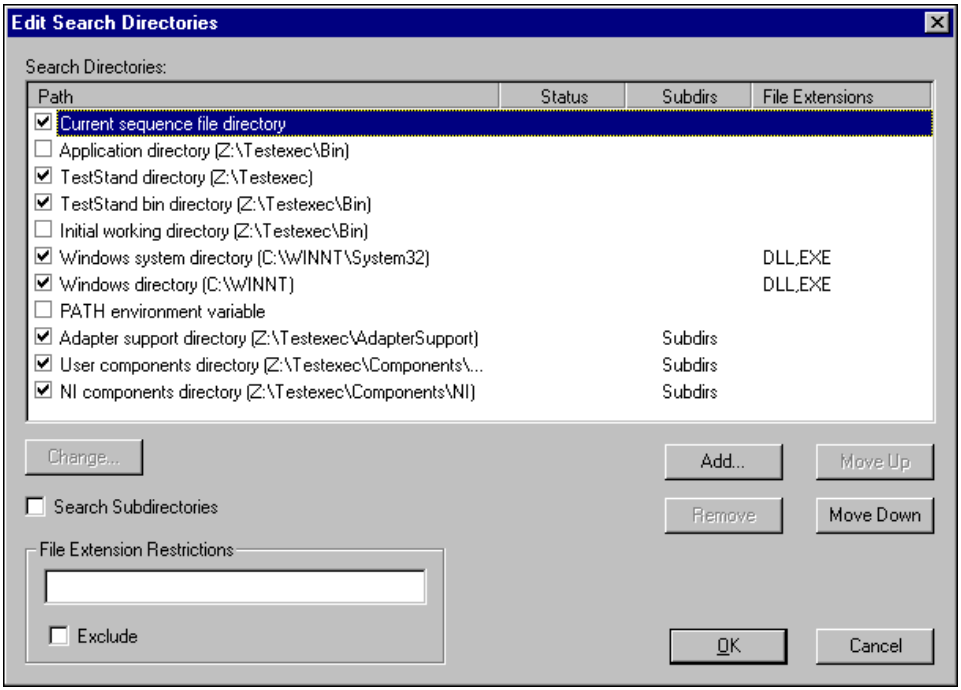

Figure 4-22. Search Directories Dialog Box

The dialog box displays a list of paths, the higher paths taking precedence over the lower paths. The list contains a default set of paths. A checkbox appears to the left of each path in the list. When you place a checkmark next to a path, TestStand includes the path in the overall search path. You can reorder paths in the list by selecting a path and clicking on the **Move Up** and **Move Down** buttons. You can add a custom directory search path with the **Add** button.

You can use the File Extension Restrictions control to search only for files with specific filename extensions. For example, to search for only DLLs and executable files, enter the following:

```
DLL, EXE
```

To search for all files except for files with specific extensions, enable the Exclude option. The dialog box prefixes the extension list with a tilde (~) in the File Extensions column.

The Search Subdirectories option specifies whether to include all subdirectories under the selected path in the overall search path.

External Viewers

The **External Viewers** command displays a dialog box in which you can specify the external viewer to use for a particular report format. You specify both the external viewer, such as Microsoft Notepad and Microsoft Internet Explorer, and the report format, such as `.txt` and `.html` files.

Use the **Add** button to add an entry to the viewer list. Use the **Delete** button to remove an entry. If you do not specify an external viewer for a format, TestStand uses the application that Windows associates with the file extension for the format.

Adapters

The **Adapters** command displays a dialog box in which you can select the active module adapter for inserting steps, or configure a specific module adapter. Refer to the *Configuring Adapters* section in Chapter 12, *Module Adapters*, for more information.

Report Options

The **Report Options** command displays a dialog box in which you can customize the generation of report files. The command calls an entry point in the default TestStand process model file. The options you set apply to all sequences you run on the station. Refer to Chapter 14, *Managing Reports*, for more information on available report options.

Tools Menu

This section explains how to use the commands in the **Tools** menu, as shown in Figure 4-23.

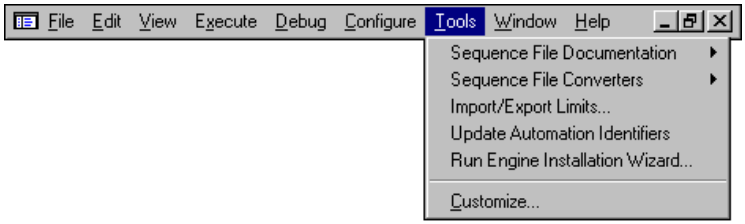

Figure 4-23. Tools Menu

Sequence File Documentation

Use the **Sequence File Documentation** submenu to generate ASCII text or HTML documentation for a sequence file.

Sequence File Converters

Use the **Sequence File Converters** submenu to convert a LabWindows/CVI or LabVIEW Test Executive sequence file into a TestStand sequence file. Refer to the *Converting From the LabVIEW Test Executive to TestStand* and *Converting From the LabWindows/CVI Test Executive to TestStand* online help documents for more information on converting sequences.

Import/Export Limits

The **Import/Export Limits** command imports step limit values from an external file or system clipboard into the steps of a sequence, or exports step limit values from steps in a sequence to an external file or system clipboard. Refer to the *Import/Export Limits Command in the Tools Menu* section in Chapter 10, *Built-In Step Types*, for more information on importing and exporting limits.

Update Automation Identifiers

If you update the interface for an ActiveX Automation server and the object and member identifiers for the server change, you must respecify any step that uses the server before you configure the ActiveX Automation Adapter to use early binding. You can use the **Update Automation Identifiers** command to update the identifiers in the active sequence file based on the name of the object or member.

For steps that create an object, the command updates the object identifiers, `CLSID` and `IID`. For steps that call a method or property, the command updates the member identifier, `MEMBERID`. Refer to the *ActiveX Automation Adapter* section of Chapter 12, *Module Adapters*, for more information on configuring the adapter to use early or late binding and on developing ActiveX servers while you are developing sequences.

Run Engine Installation Wizard

Use the **Run Engine Installation Wizard** command to create a custom TestStand engine installation. Refer to the *Creating a Run-Time TestStand Engine Installation* section in Chapter 16, *Distributing TestStand*, for more information on using the installation wizard.

Customize

Use the **Customize** command to create your own entries in the **Tools** menu. Figure 4-24 shows the Customize Tool Menu dialog box.

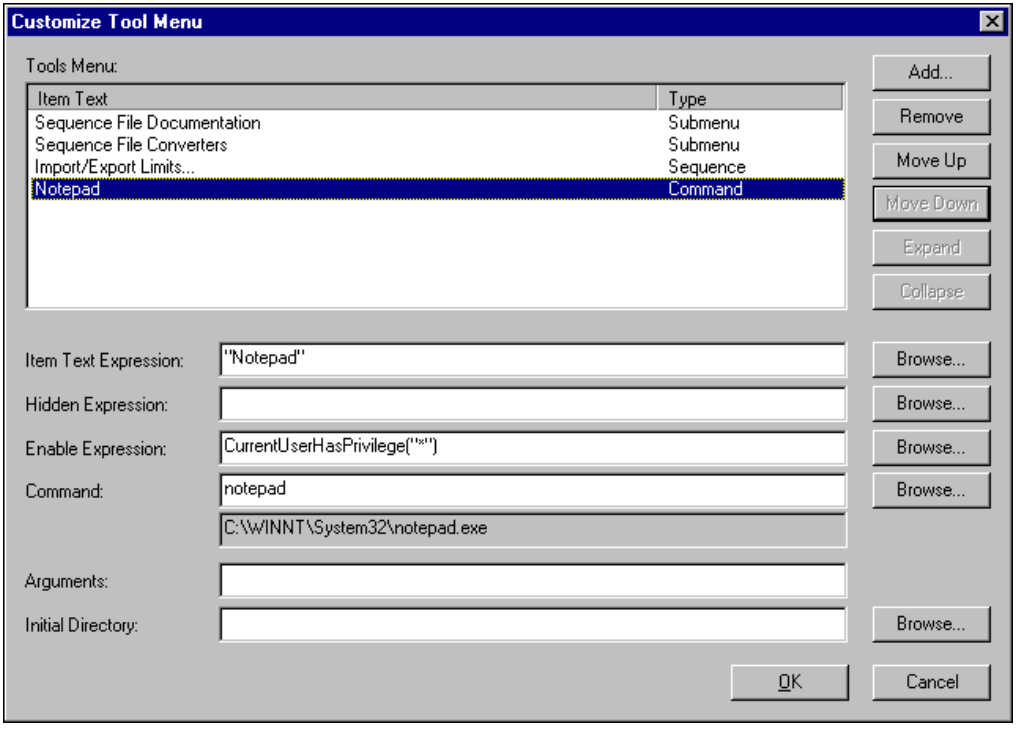

Figure 4-24. Customize Tool Menu Dialog Box

Use the **Add** button to insert a new menu item above the selected item in the Tools Menu list. You can add the following types of menu items.

- **Submenu**—Contains additional menu items.
- **Command**—Invokes a Windows executable.
- **Sequence**—Initiates an execution on a sequence in a sequence file.
- **Sequence File**—Creates a submenu that lists all sequences in a sequence file as menu items.

The **Remove** button deletes the menu item from the **Tools** menu. Use the **Move Up** and **Move Down** buttons to change the order of items within the menu or submenus.

Use the **Expand** button to view the menu items in a submenu. Use **Collapse** to step out of a items in a submenu.

The Item Text Expression option specifies the literal text to display for the menu item. You can specify an expression that evaluates to the literal text.

The Hidden Expression option allows you to specify an expression that determines when the menu item is hidden. If empty, the expression is assumed to be `False`.

Note ***TestStand evaluates the Hidden Expression in the*** `ConstructToolMenus` ***method of the*** `Engine` ***class. The sequence editor constructs the*** **Tools** ***menu by calling this method each time you click on it, but the operator interfaces construct the*** **Tools** ***menu only once during initialization.***

The Enable Expression option appears for Command and Sequence item types. It lets you specify an expression that determines when the menu item is enabled. The expression must return `True` to enable the menu item and `False` to dim the menu item.

The Command, Arguments, and Initial Directory options appear only for the Command menu type. The Command option specifies the executable path. The Arguments option specifies the command line arguments. The Initial Directory option specifies the initial working directory for the executable.

The Sequence File and Sequence options specify the target for the Sequence File and Sequence menu item types.

Window Menu

This section explains how to use the commands in the **Window** menu, as shown in Figure 4-25.

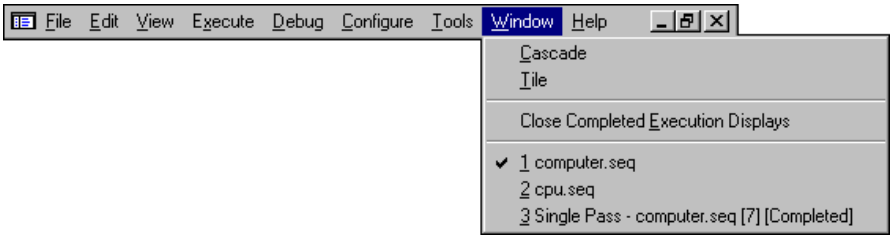

Figure 4-25. Window Menu

Cascade

Use the **Cascade** command to arrange all open windows so that each title bar is visible.

Tile

Use the **Tile** command to arrange all open windows in smaller sizes to fit next to each other.

Close Completed Execution Displays

Use the **Close Completed Execution Displays** command to close all execution displays that are no longer executing.

Open Windows

A list of all open windows appears at the bottom of the **Window** menu.

5

Sequence Files

This chapter describes TestStand sequence files. Each sequence file contains one or more sequences. Sequences, in turn, contain steps that conduct tests, set up instruments, or perform other actions necessary to test a UUT. In addition to sequences, sequence files can contain global variables. You can access sequence file global variables from every sequence in the file. Sequence files also contain the definitions for the data types and step types that the sequences in the file use.

You can use the sequence editor to create and edit sequence files. You can execute sequences from the sequence editor or from any other TestStand operator interface program.

There are multiple types of sequence files. Most sequence files you work with are *normal sequence files*. Normal sequence files contain sequences that test UUTs. *Model sequence files* contain process model sequences. *Station Callback sequence files* contain the station callback sequences. *Front-End Callback sequence files* contain Front-End callback sequences. Usually, your computer has only one Station Callback sequence file and one Front-End Callback sequence file.

In the sequence editor, you use a Sequence File window to view and edit a sequence file. You can open an existing sequence file into a Sequence File window by selecting **File»Open**. You can create a new Sequence File Window by selecting **File»New**.

Sequence File Window Views

You use the View ring at the top right of the Sequence File window to select the aspect of the file to display. You can use the View ring to view an individual sequence, a list of all sequences in the file, the global variables in the file, or the types that you use in the file.

Figure 5-1 shows the contents of the View ring for an example sequence file.

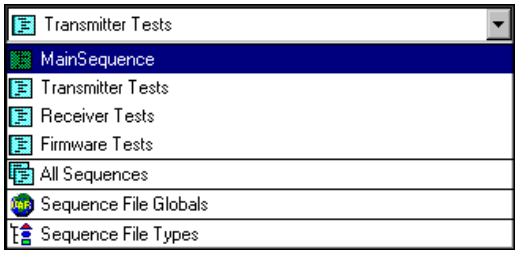

Figure 5-1. Sequence File View Ring

All Sequences View

Sequence files can contain multiple sequences. You can display a list of the sequences in a file by selecting All Sequences from the View ring. You can use this view to create new sequences and to cut, copy, and paste sequences. You also can drag and drop sequences to the All Sequences view in another Sequence File window.

Figure 5-2 shows the All Sequences view for an example file.

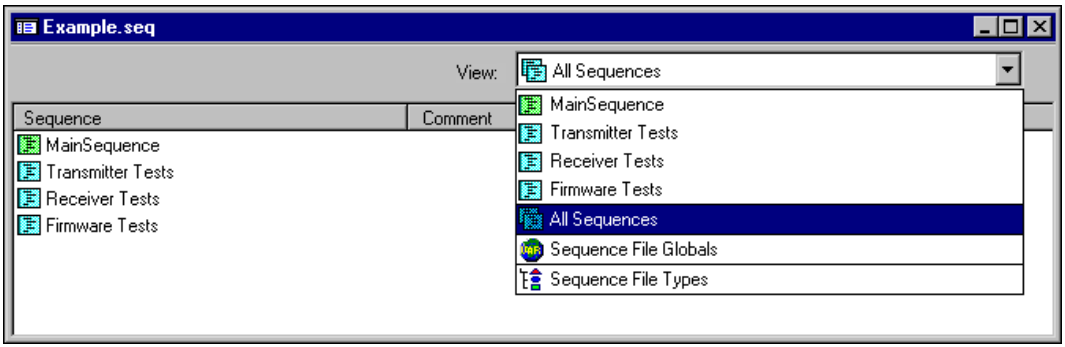

Figure 5-2. All Sequences View in the Sequence File Window

Sequence View Context Menu

You can display a context menu by right-clicking on the view. The items in the context menu vary depending on the whether you right-click on a sequence or on the background area of the view. The context menu can contain the following items.

Open Sequence

The **Open Sequence** command changes the sequence file view to display the contents of the selected sequence.

Insert Sequence

The **Insert Sequence** command adds a new sequence to the sequence file.

Rename

The **Rename** command allows you to edit the name of the selected sequence

Browse Sequence Context

The **Browse Sequence Context** command displays a tree view that contains the names of variables, properties, and sequence parameters you can access from expressions and step modules when the sequence is running. This command also appears in the **View** menu of the sequence editor menu bar. Refer to the *View Menu* section in Chapter 4, *Sequence Editor Menu Bar*, for more information.

View Contents

The **View Contents** command changes the sequence file view to display the contents of the selected sequence.

Sequence Properties

The **Properties** command displays the Sequence Properties dialog box for the selected sequence. You use the Sequence Properties dialog box to view and edit the built-in properties of the selected sequence. Usually, the dialog box has a single tab titled General. If the current sequence file is a process model file, the dialog box has a second tab titled Model.

Figure 5-3 shows the Sequence Properties dialog box.

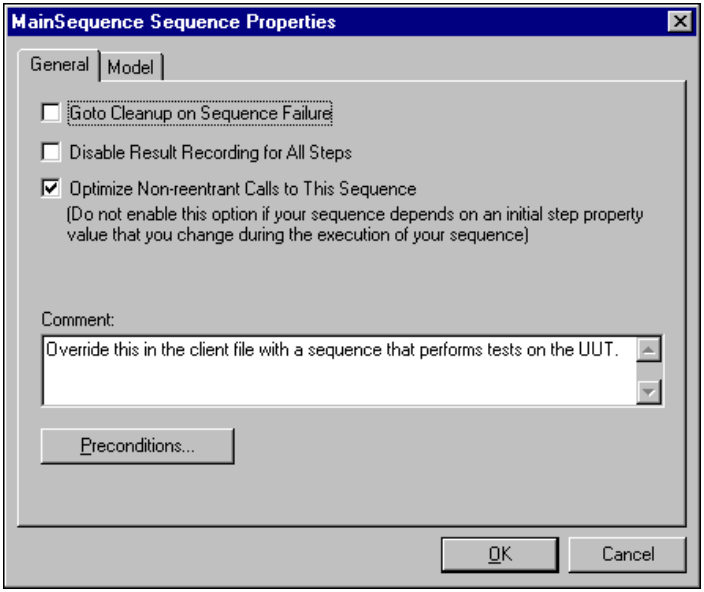

Figure 5-3. Sequence Properties Dialog Box

The General tab contains the following controls:

- **Goto Cleanup on Sequence Failure**—TestStand maintains an internal status value for each executing sequence. When the status property of a step is set to `Failed` and the Step Failure Causes Sequence Failure option is enabled for the step, TestStand sets the internal sequence status value to `Failed`. The Goto Cleanup on Sequence Failure option controls the flow of execution when the internal sequence status value is set to `Failed`. Enable this option if you want the execution to branch immediately to the Cleanup step group. Disable this option if you want execution to continue normally at the next step.
- **Disable Results for All Steps**—This option prevents TestStand from adding results for the steps in the sequence to the results list. Refer to the *Result Collection* section in Chapter 6, *Sequence Execution*, for more information on the results list.

- **Optimize Non-Reentrant Calls to this Sequence**—This option decreases the time it takes TestStand to call the sequence after the first call to the sequence in an execution. If this option is disabled, TestStand initializes a new copy of each custom step property in a sequence each time it calls the sequence. TestStand does this so that the sequence always begins executing with the initial property values that the steps in the sequence specify. This initialization is necessary only if a sequence relies on the initial value of a custom step property and then modifies its value. Few sequences do this.

  When you enable this option, TestStand initializes the values of custom step properties in the sequence the first time it calls the sequence in an execution. TestStand saves the values of the custom step properties after the sequence completes and reuses the values when it calls the sequence again. If the same sequence is called at the same time in different threads or recursively within the same thread, TestStand creates unique copies of the custom step properties.
- **Comment**—You can use this control to place a comment for the sequence in the All Sequences View. The sequence comment also appears in the documentation that TestStand generates for the sequence file.
- **Preconditions**—This button displays the Preconditions dialog box. You can use the Preconditions dialog box to specify the conditions that must be true for each step in the sequence to run. When you access this dialog box from the Step Properties dialog box, it applies only to a particular step. When you access the dialog box from the Sequence Properties dialog box, you can view and edit the preconditions for each step in the sequence. Refer to the *Preconditions Dialog Box* section later in this chapter for more information.

If the sequence file is a process model file, the Sequence Properties dialog box also has a Model tab. Refer to Chapter 13, *Process Models*, for more information on sequence properties that are unique to process model files.

Sequence File Properties

The **Sequence File Properties** command displays the Sequence File Properties dialog box for the sequence file. Figure 5-4 shows the General tab on the Sequence File Properties dialog box.

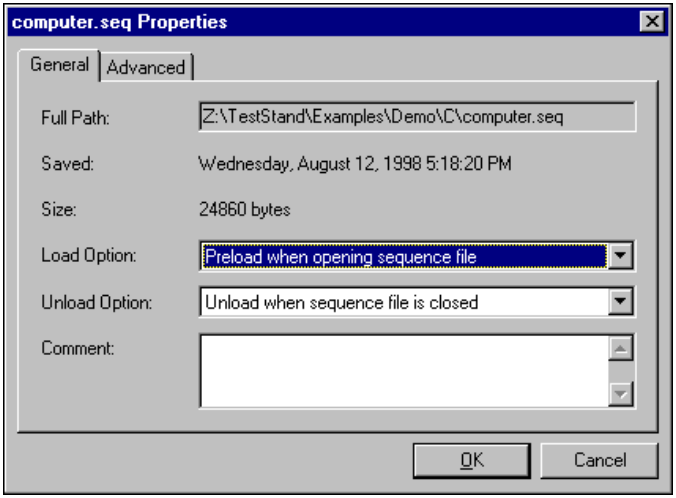

Figure 5-4. General Tab on the Sequence File Properties Dialog Box

The General tab contains the following controls:

- **Full Path**—This control displays the location of the sequence file on disk.
- **Saved**—This control displays the time at which you last saved the sequence file.
- **Size**—This control displays the size of the sequence file on your disk drive.

- **Load Option**—You can use this control to specify a load option setting for every step in the sequence file. The choices are:
  - Preload when opening sequence file
  - Preload when execution begins
  - Load dynamically
  - Use step load option

    The Use Step Load Option setting tells TestStand to load each code module according to the load option for the particular step that calls the code module. Refer to the *Step Properties Dialog Box* section, later in this chapter, for more information on the other load option values.
- **Unload Option**—You can use this control to specify an Unload Option setting for every step in the sequence file. The choices are:
  - Unload when precondition fails
  - Unload after step executes
  - Unload after sequence executes
  - Unload when sequence file is closed
  - Use step unload option

    The Use Step Unload Option setting tells TestStand to unload each code module according to the unload option for the particular step that call the code module. Refer to the *Step Properties Dialog Box* section, later in this chapter, for more information on the other unload option values.

Note ***If you enable the sequence property, Optimize Non-Reentrant Calls to This Sequence, TestStand does not unload the code modules for the sequence until after the execution ends, regardless of the unload options for the sequence file or the steps in the sequence.***

- **Comment**—You can use this control to place a comment that appears in the documentation that TestStand generates for the sequence file.

Figure 5-5 shows the Advanced tab on the Sequence File Properties dialog box.

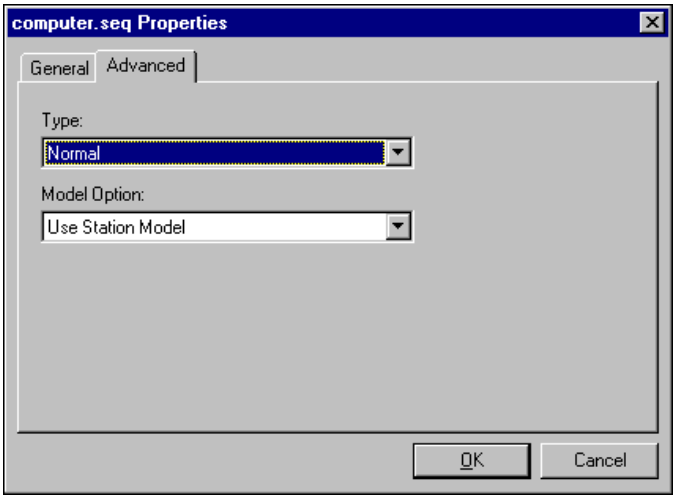

Figure 5-5. Advanced Tab on the Sequence File Properties Dialog Box

The Advanced tab contains the following controls:

- **Type**—You can use the Type control to specify the type setting for the sequence file. The possible settings are:
  - Normal
  - Model
  - Front-End Callbacks
  - Station Callbacks
  - Reserved

  If you use the sequence file as a process model, set the type to Model. If you do not use the file as a process model, leave the type set to Normal.
- **Model Option**—Use this option to select the process model file to use for the sequence file. The possible settings are:
  - Use Station Model—Select this value to use the process model file that the Station Model option in the Station Options dialog box specifies. This is the default setting for this option.
  - No Model—Select this value to specify that the sequence file does not use a process model.

– Require Specific Model—Select this value to specify a particular process model file. If you select this value, the tab displays additional controls that you can use to specify the location of a process model file.

Refer to the *Process Models* section in Chapter 1, *TestStand Architecture Overview*, and to Chapter 13, *Process Models*, for more information on process models.

Sequence File Callbacks

The **Sequence File Callbacks** command displays the Callbacks dialog box for the sequence file. Figure 5-6 shows the Callbacks dialog box.

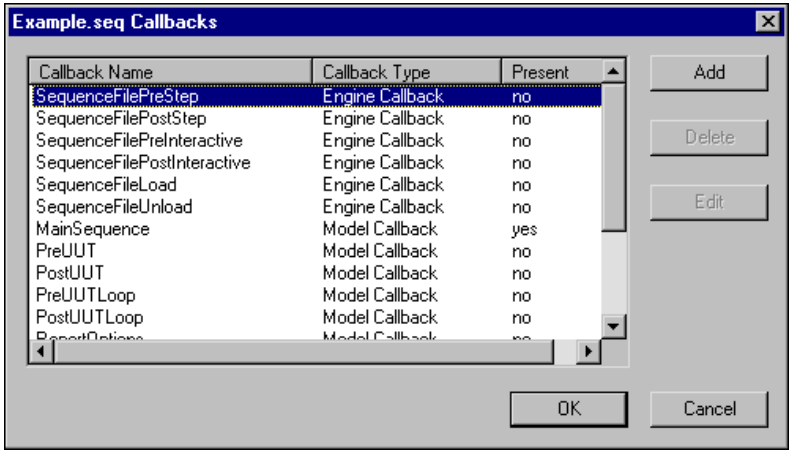

Figure 5-6. Callbacks Dialog Box

The Callbacks dialog box lists every callback that you can override in the sequence file. The columns in the list display the name of the callback, indicate whether the callback is an Engine or Model callback, and indicate whether the sequence file overrides the callback. You use the **Add** button to override the selected callback by inserting a sequence with the same name into the sequence file. You can use the **Delete** button to delete the sequence that overrides the selected callback. You can use the **Edit** button to dismiss the dialog box and display the sequence that overrides the selected callback. Refer to the *Process Models* section in Chapter 1, *TestStand Architecture Overview*, for more information on model callbacks. Refer to the *Engine Callbacks* section in Chapter 6, *Sequence Execution*, for more information on engine callbacks.

The following restrictions apply to the `SequenceFileLoad` and `SequenceFileUnload` callbacks.

- TestStand can deadlock when it executes a `SequenceFileLoad` callback that calls into another sequence file containing a `SequenceFileLoad` callback which calls back into the original sequence file. This can occur with any number of levels of sequence files as long as the dependencies among the `SequenceFileLoad` callbacks exist between sequence files.
- TestStand can enter an infinite loop when it executes a `SequenceFileUnload` callback that calls into another sequence file containing a `SequenceFileUnload` callback which calls back into the original sequence file. The infinite loop can be broken by selecting the **Debug»Terminate All Executions** command.
- You must not define a `SequenceFileUnload` callback in the `FrontEndCallback.seq` or `StationCallbacks.seq` sequence files. If you do this, TestStand hangs when you shut down the TestStand engine.

Individual Sequence View

Each sequence can contain steps, parameters, and local variables. You can view the contents of a specific sequence by selecting it from the View ring. Figure 5-7 shows the contents of an example sequence.

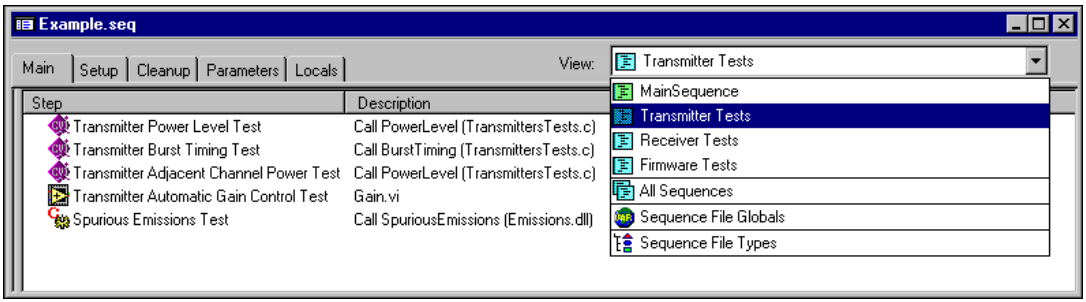

Figure 5-7. Individual Sequence View for an Example Sequence

The Sequence view has five tabs: Main, Setup, Cleanup, Parameters, and Locals. You select a tab to choose which part of the sequence to view.

Main, Setup, and Cleanup Tabs

The Main, Setup, and Cleanup tabs each show one of the step groups in the sequence. In the Setup step group, insert steps that initialize or configure your instruments, fixtures, and UUT. In the Main step group, insert steps that test your UUT. In the Cleanup step group, insert steps that power down or de-initialize your instruments, fixtures, and UUT. Refer to Chapter 6, *Sequence Execution*, for more information on how TestStand uses the different step groups.

Step Group List View and Tree View

Each step group tab normally displays a list of the steps in the group. This list is called the *step group list view*. You can drag the *step group divider bar* away from the left edge of the window to reveal a tree-structured view that allows you to browse the custom properties for each step. This tree-structured view is called the step group *tree view*. The list view always displays the contents of the item that you select in the tree view. Usually, you only use the tree view when you design and debug a new step type. In Figure 5-8, the tree view shows the custom properties of a Numeric Limit Test step.

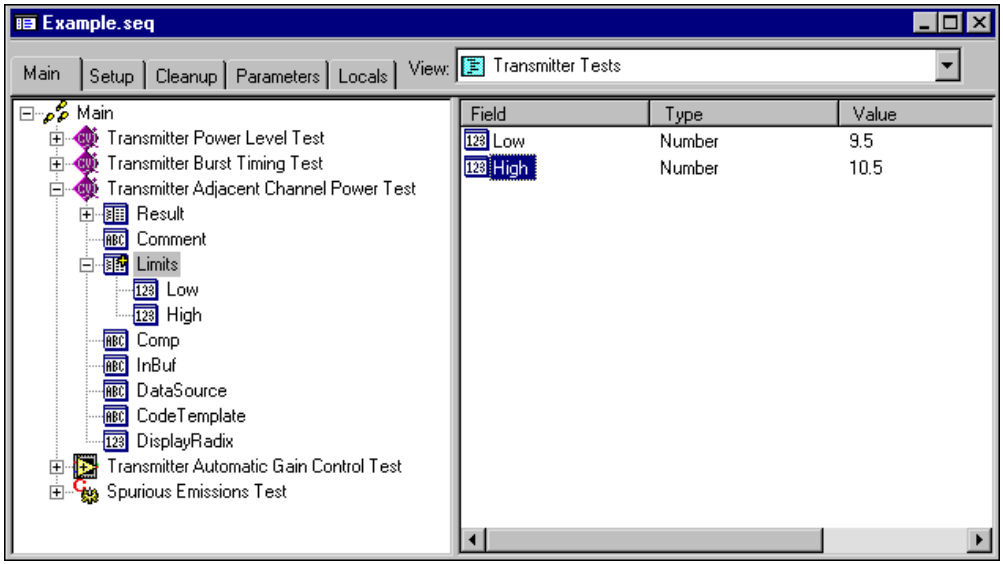

Figure 5-8. The Step Group Tree View (Left) and List View (Right)

Step Group List View Columns

The columns in the list view for a step group vary according to whether the list view is displaying steps or step properties. Figure 5-9 shows the list view displaying steps.

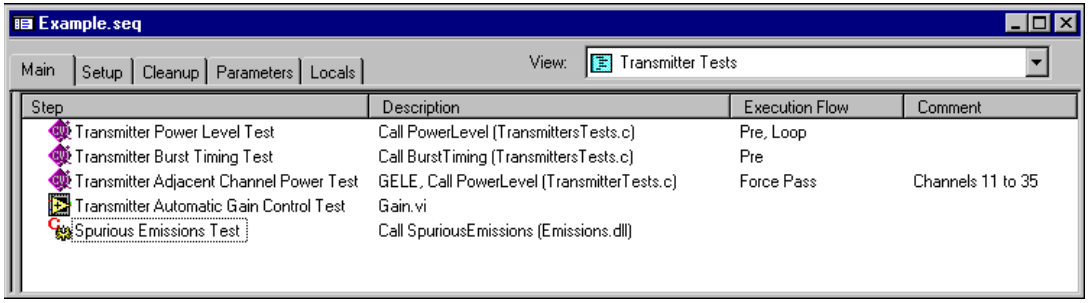

Figure 5-9. Step Group List View Columns for Steps

When the list view displays steps, it contains the following columns:

- **Step**—This column displays the name of the step and its icon. You can click to the left of the step icon to toggle the breakpoint for the step.
- **Description**—This column displays a description of the step that varies according to the type of step and the adapter with which it was created.
- **Execution Flow**—This column indicates whether the properties of the step are set to control the flow of execution in the sequence. The values that can appear in this column are:
  - Pre—Indicates that the step has a precondition.
  - Post—Indicates that the step has a post action.
  - Loop—Indicates that step has been configured to loop.
  - Skip—Indicates that the run mode of the step has been set to `SKIP`.
  - Force Pass—Indicates that the run mode of the step has been set to `PASS`.
  - Force Fail—Indicates that the run mode of the step has been set to `FAIL`.
- **Comment**—This column displays the comment for the step that you specify in the Step Properties dialog box.

Figure 5-10 shows the list view displaying step properties.

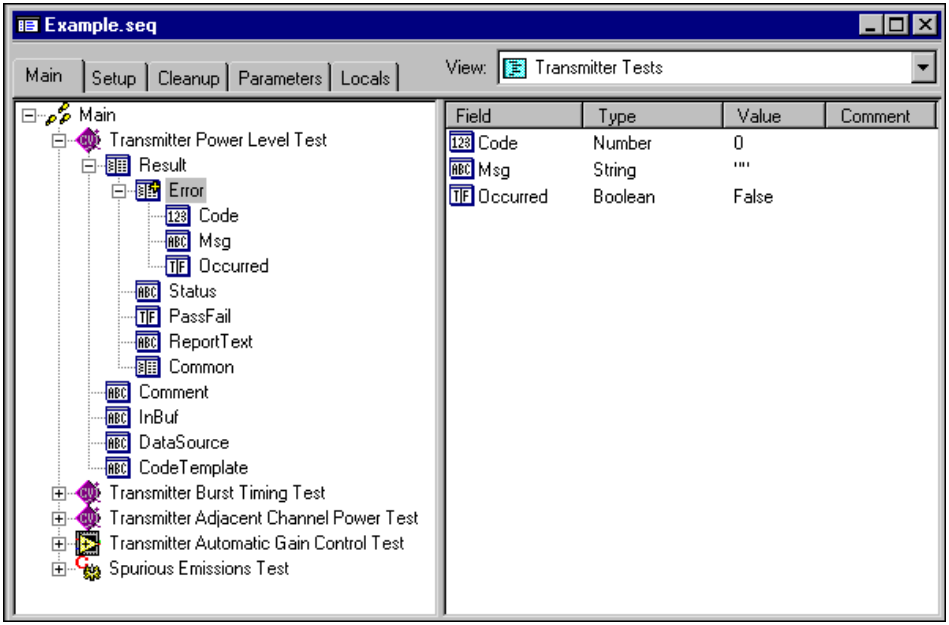

Figure 5-10. Step Group List View Columns for Step Properties

When the list view displays step properties, it contains the following columns:

- **Field**—This column displays the names and icons for the subproperties of the step property that is currently selected in the tree view. If the property selected in the tree view contains a value or array of values, the value names and icons also appear in this column.
- **Type**—This column displays the data type of each subproperty or value.
- **Value**—This column displays the current value for each subproperty or value element.
- **Comment**—If a subproperty in the list view has a subproperty of its own named `Comment`, the value of the `Comment` subproperty appears in this column.

Note ***You can expand a column to the width of its largest entry by double-clicking on the vertical separator at the right edge of the column heading. This is especially useful when an item has a long comment.***

Step Group Context Menu

You can display a context menu by right-clicking on the tree view or list view. The items in the context menu vary depending on the whether you right-click on a step, step property, the background area of the tree view, or the background of the list view. The context menu can contain the following items.

Insert Step

The **Insert Step** menu item has a submenu in which you select the type of step you want to insert into the sequence. Figure 5-11 shows the **Insert Step** submenu.

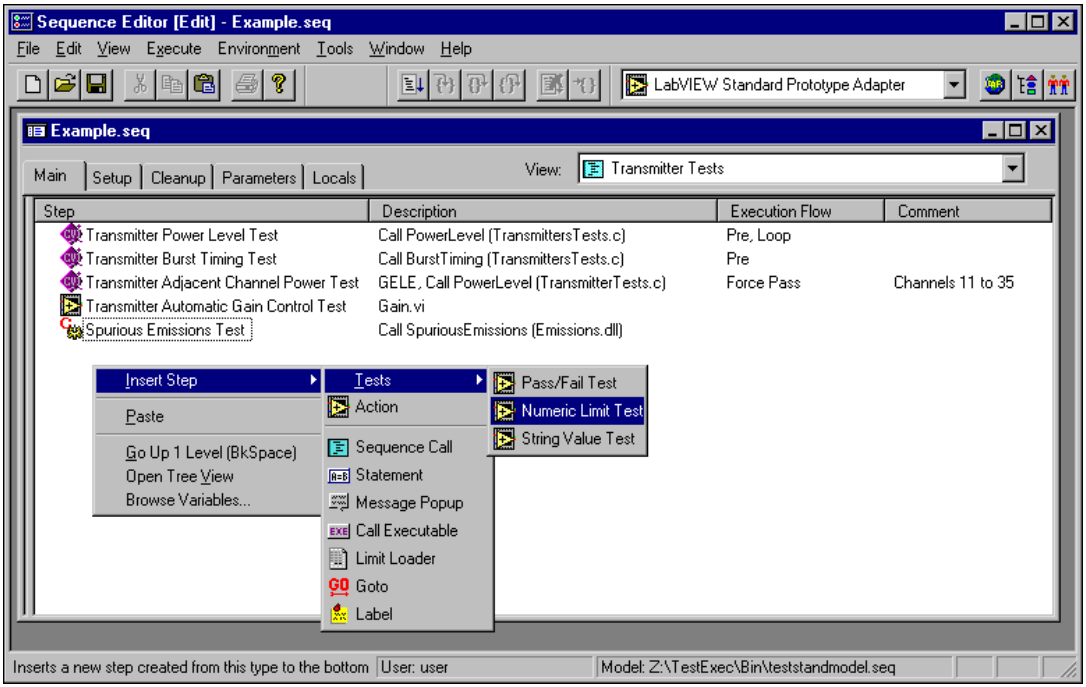

Figure 5-11. Insert Step Menu with LabVIEW Standard Prototype Adapter Selected

Many of the steps types in the **Insert Step** submenu allow you to call code modules. Of these step types, some can work with all module adapters, while others require a specific module adapter. Each adapter allows you to call a category of code modules, such as LabVIEW VIs, LabWindows/CVI source or object modules, or DLLs. Some adapters also know how to control the application development environments in which you build these types of code modules.

Before you insert a step that can call a code module using any adapter, you must select the appropriate adapter for the type of code module you want the step to call. You use the pull down ring in the sequence editor tool bar to select a module adapter. The pull down ring shows an icon for each adapter. The icon for the currently selected adapter appears next to all step types in the **Insert Step** submenu that can work with any adapter. After you insert a step, the adapter icon also appears next to the step name. After you create a step with a particular module adapter, you cannot change its adapter.

Each step type that requires a particular module adapter appears in the **Insert Step** submenu next to the icon for that adapter. After you insert a step using one of these step types, the adapter icon appears next to the step name.

When you insert a step that does not call a code module, such as a Goto or Label step, the currently selected adapter has no effect. These step types have their own icons. The icons appear next to these step types in the **Insert Step** submenu and next to the steps that you create using these step types.

Edit

The menu item name of the **Edit** command varies according to the type of the selected step. For example, the menu item name is **Edit Message Settings** for a Message Popup step, **Edit Limits** for a Numeric List Test step, and **Edit Destination** for a Goto step. For each step, the menu item displays a dialog box that you use to edit the settings that are unique to the type of the step. Some step types, such as the Label step and the Action step, do not have step-type-specific settings. For these step types, this menu item is disabled.

Specify Module

The **Specify Module** command displays a Specify Module dialog box for the selected step. The dialog box that appears depends on the module adapter for the step. You use the Specify Module dialog box to specify the code module that the step calls. You also can specify options that TestStand uses when it calls the step. Refer to Chapter 12, *Module Adapters*, for more information on the Specify Module dialog box for each adapter.

Edit Code

The **Edit Code** command displays the source code for the code module that the step calls. TestStand uses the module adapter for the step to determine the appropriate application in which to display the source code.

Toggle Breakpoint

The **Toggle Breakpoint** command sets or clears the breakpoint state for the selected steps.

Run Mode

The **Run Mode** menu item displays a submenu from which you can set the following run mode values for the selected steps.

- **Force Pass**—TestStand does not execute the step and does not evaluate its preconditions. Instead, TestStand sets the status of the step to `Passed` automatically.
- **Force Fail**—TestStand does not execute the step and does not evaluate its preconditions. Instead, TestStand sets the status of the step to `Failed` automatically.
- **Skip**—TestStand does not execute the step and does not evaluate its preconditions. Instead, TestStand sets the status of the step to `Skipped` automatically.
- **Normal**—This value tells TestStand to execute the step normally. This is the default value.

Run Selected Steps

The **Run Selected Steps** command runs the selected steps in interactive mode. Refer to the *Interactively Executing Steps* section in Chapter 6, *Sequence Execution*, for more information on running steps in interactive mode.

Loop Selected Steps

The **Loop Selected Steps** command loops on the selected steps in interactive mode. Before running the steps, this command displays a dialog box that you use to specify how many times to loop. Refer to the *Interactively Executing Steps* section in Chapter 6, *Sequence Execution*, for more information on running steps in interactive mode.

Open Tree View

The **Open Tree View** command moves the step group divider bar away from the left edge of the window so that the tree view is visible. You can use the tree view to browse the custom properties contained in each step.

Close Tree View

The **Close Tree View** command hides the tree view by moving the step group divider bar flush against the left edge of the window. The command also causes the list view to display the steps in the step group.

View Contents

The **View Contents** command selects the tree view node that corresponds to the currently selected item in the list view. The list view then displays the contents of the item. If the tree view is currently closed, it opens to show the selected node. You use this command on steps or properties to view their subproperties.

Go Up One Level

The **Go Up One Level** command selects the next higher level node in the tree view. The list view displays the contents of the newly selected node. If you invoke this command when the highest level node is selected in the tree view, the Sequence File window displays the All Sequences view.

Browse Sequence Context

The **Browse Sequence Context** command displays a tree view that contains the names of variables, sequence parameters, and step properties you can access from expressions and step modules when the selected step is running. This command also appears in the **View** menu of the sequence editor menu bar. Refer to the *View Menu* section in Chapter 4, *Sequence Editor Menu Bar*, for more information.

Sequence Properties

The **Sequence Properties** command displays the Sequence Properties dialog box. Refer to the *Sequence View Context Menu* section earlier in this chapter for more information on the Sequence Properties dialog box.

Step Properties Dialog Box

The **Properties** command displays the Properties dialog box for the selected step or property. If the selected item is a property, you can use the Properties dialog box to edit the value of the property. If the selected item is a step, the Step Properties dialog box appears.

Figure 5-12 shows the General tab on the Step Properties dialog box.

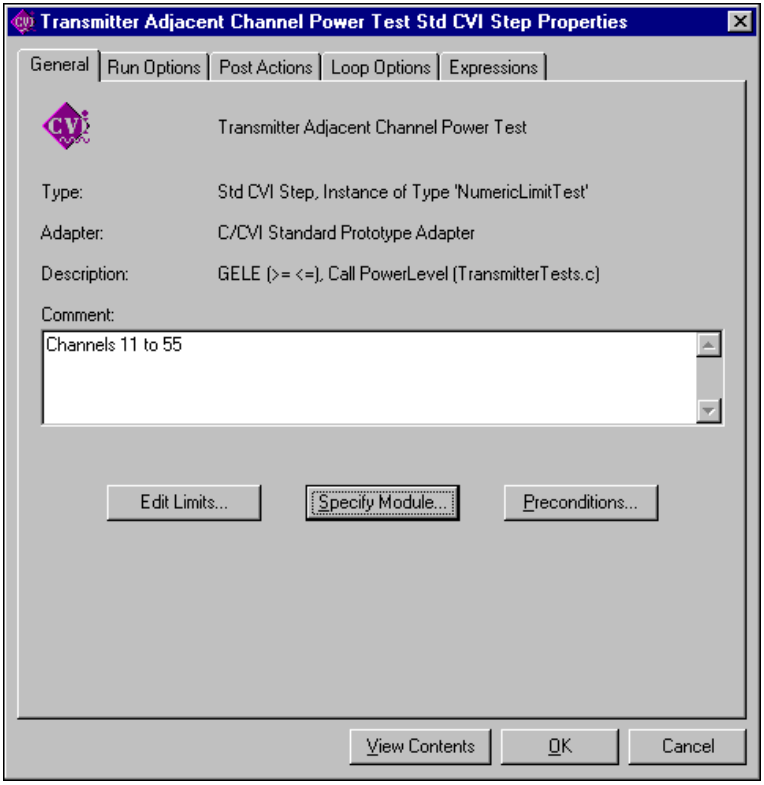

Figure 5-12. General Tab on the Step Properties Dialog Box

The General tab on the Step Properties dialog box contains the following controls:

- **Comment**—You can use this control to place a comment for the step in the Step Group list view. The step comment also appears in the documentation that TestStand generates for the sequence file.
- **Edit**—Displays a dialog box you use to edit the settings that are unique to the type of the step. The button caption varies according to the type of the step. For more information, refer to the discussion of the **Edit** context menu item earlier in this section.
- **Specify Module**—Displays the Specify Module dialog box for the selected step. The dialog box that appears depends on the module adapter for the step. You use the Specify Module dialog box to specify the code module that the step calls. You also can specify options that TestStand uses when it calls the step. Refer to Chapter 12, *Module*

Adapters, for more information on the Specify Module dialog box for each adapter.

- **Preconditions**—Displays the Preconditions dialog box. You can use the Preconditions dialog box to specify the conditions that must be true for the step to execute. Refer to the *Preconditions Dialog Box* section later in this chapter for more information.

Figure 5-13 shows the Run Options tab on the Step Properties dialog box.

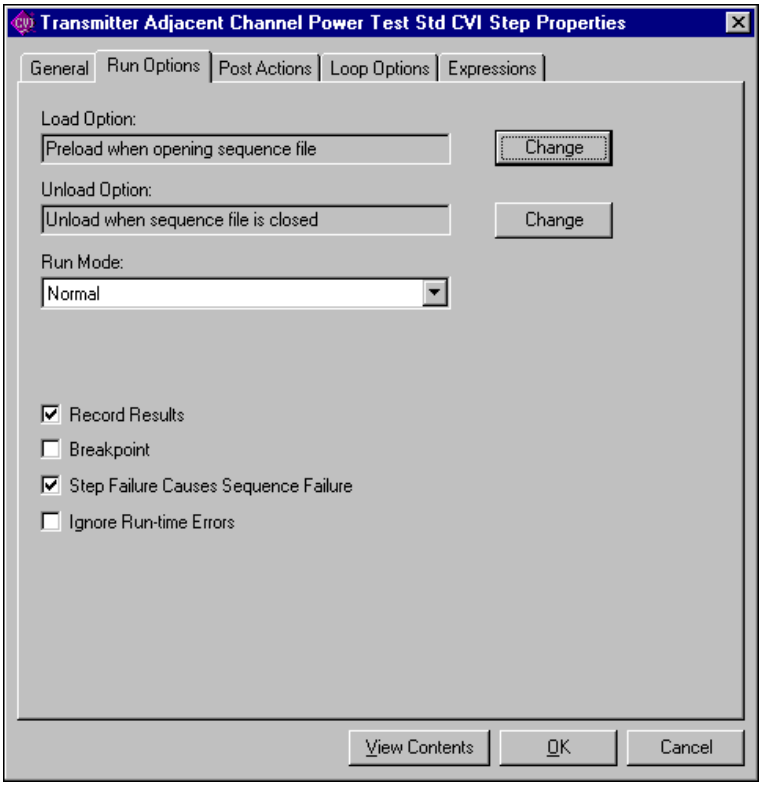

Figure 5-13. Run Options Tab on the Step Properties Dialog Box

The Run Options tab on the Step Properties dialog box contains the following controls:

- **Load Option**—You can use this control to specify a load option setting for the step. The choices are:
  - Preload when opening sequence file—TestStand loads the step module when TestStand loads into memory the sequence that contains the step.
  - Preload when execution begins—TestStand loads the step module when any sequence in the sequence file that contains the step begins executing. This value is the default setting.
  - Load dynamically—TestStand does not load the step module until the step is ready to call it.
- **Unload Option**—You can use this control to specify an Unload Option setting for the step. The choices are:
  - Unload when precondition fails—TestStand unloads the step module when the precondition for the step evaluates to `False`.
  - Unload after step executes—TestStand unloads the step module after the step finishes executing.
  - Unload after sequence executes—TestStand unloads the step module after the sequence that contains it finishes executing.
  - Unload when sequence file is closed—TestStand unloads the step module when TestStand unloads the sequence file that contains the step from memory. This value is the default setting.

Note ***If you enable the sequence property, Optimize Non-Reentrant Calls to This Sequence, TestStand does not unload the code modules for the sequence until after the execution ends, regardless of the unload options for the sequence file or the steps in the sequence.***

- **Run Mode**—Use this ring to set the following run mode values for the step.
  - Force Pass
  - Force Fail
  - Skip
  - Normal
- **Record Results**—This option determines whether the contents of the `Result` property for the step are added to the result list for the sequence. Refer to the *Result Collection* section in Chapter 6, *Sequence Execution*, for more information on result collection.

- **Breakpoint**—This option tells TestStand to break at this step before executing it. You also can set the breakpoint state for a step by selecting the **Toggle Breakpoint** item in the context menu or by clicking to the left of the step icon.
- **Step Failure Causes Sequence Failure**—TestStand maintains an internal status value for each executing sequence. When the status property of a step is set to `Failed`, and the Step Failure Causes Sequence Failure option is enabled for the step, TestStand sets the internal sequence status value to `Failed`. If the internal status of the sequence is `Failed` when the sequence returns, TestStand sets the status of the calling step to `Failed`. This affects steps that use the Action or Sequence Call step types. Steps that use the Pass/Fail Test, Numeric Limit Test, and String Value Test step types overwrite the step status.
- **Ignore Run-time Errors**—This option prevents the step from reporting a run-time error to the sequence. When a step causes a run-time error, the step stops executing, and TestStand sets the status of the step to `Error`. If this option is disabled, TestStand also sets the internal status of the sequence to `Error`, and execution branches to the Cleanup step group for the sequence. If this option is enabled, TestStand does not set the internal status of the sequence to `Error`. Instead, TestStand resets the `Error.Occurred` property of the step to `False` and execution continues normally with the next step. The value of the `Result.Status` property remains set to `Error` for the step.

If the step is a sequence call, the Run Options tab displays two addition controls:

- **Sequence Call Trace Setting**—You can use this option to control tracing when calling a subsequence. The possible values are:
  - Use current trace setting—TestStand does change the current tracing state when it calls the subsequence. This is the default value. Usually, only process model files use other values for this option.
  - Enable tracing in sequence—TestStand enables tracing when it calls the subsequence, and it restores the original tracing state when the subsequence returns.
  - Disable tracing in sequence—TestStand disables tracing when it calls the subsequence, and it restores the original tracing state when the subsequence returns. However, if you enable the Allow Tracing into Sequence Calls Marked with Tracing Off option in the Station Options dialog box, TestStand ignores this setting and does not alter the tracing state when it calls the subsequence.

- **Ignore Termination**—This option controls what happens when a subsequence that you call from this step causes execution to terminate. If you enable this option, TestStand terminates the subsequence, sets the status of the calling step to `Terminated`, but allows the calling sequence to proceed normally from the next step. Usually, only process model files use this option. This option has no effect when execution aborts. Refer to the *Terminating and Aborting Executions* section in Chapter 1, *TestStand Architecture Overview*, for more information on execution termination.

You can use the Post Actions tab on the Step Properties dialog box to specify an action that occurs after the step executes. You can make the action conditional on the Pass/Fail status of the step or on any custom condition.

Figure 5-14 shows the Post Actions tab on the Step Properties dialog box.

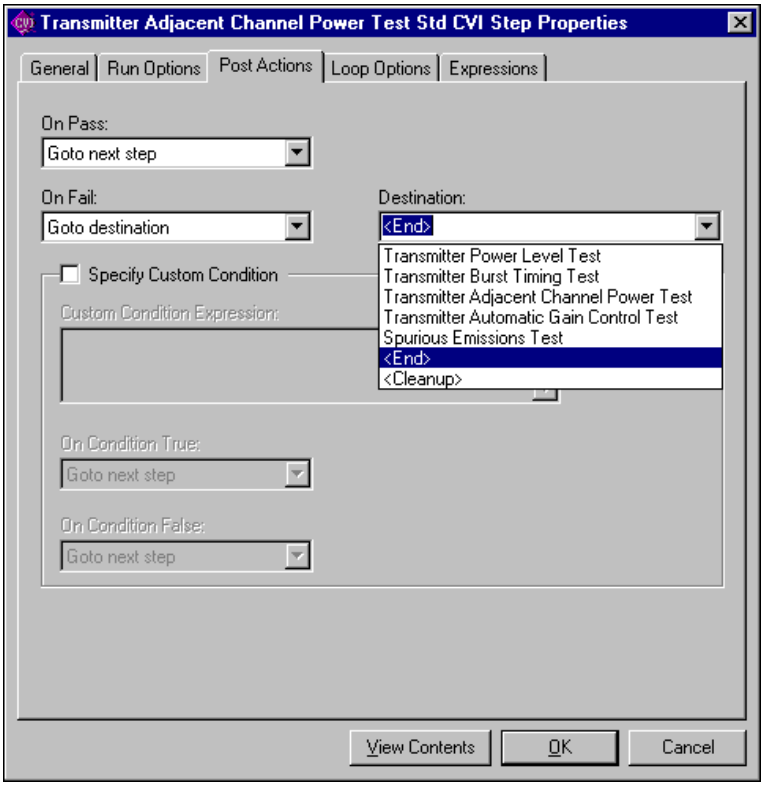

Figure 5-14. Post Actions Tab on the Step Properties Dialog Box

The Post Actions tab on the Step Properties dialog box contains the following controls:

- **On Pass**—You can use this control to specify an action that occurs when the step completes and its status is `Passed`.
- **On Fail**—You can use this control to specify an action that occurs when the step completes and its status is `Failed`.
- **Destination**—Specifies the destination step for the On Pass, On Fail, On Condition True, and On Condition False controls.
- **Specify Custom Condition**—This option enables you to specify a custom condition to control the post action for the step.
- **Custom Condition Expression**—Use this control to specify the custom Boolean expression that controls the post action for the step.
- **On Condition True**—Use this control to specify the action that occurs when the step completes and the custom condition expression evaluates to `True`.
- **On Condition False**—You can use this control to specify the action that occurs when the step completes and the custom condition expression evaluates to `False`.

  The On Pass, On Fail, On Condition True, and On Condition False controls give you the following post actions to select from.

  – Goto next step—Execution continues normally with the next step. This is the default value.

  – Goto destination—Execution branches to the destination you select. You can branch to any step in the current step group, to the end of the current step group, or to the Cleanup step group. If the post action for a step specifies that execution branches to the Cleanup step group and the current step is in the Cleanup step group, execution proceeds normally with the next step in the Cleanup group.

  – Terminate execution—Execution terminates. Refer to the *Terminating and Aborting Executions* section in Chapter 1, *TestStand Architecture Overview*, for more information on execution termination.

  – Call sequence—TestStand calls a sequence before continuing to the next step. You can select any sequence in the sequence file. TestStand does not pass any arguments to the sequence. If the sequence has parameters, TestStand uses their default values.

  – Break—TestStand breakpoints before continuing to the next step.

You can use the Loop Options tab on the Step Properties dialog box to configure an individual step to run repeatedly in a loop when it executes. If you want to loop on several steps at once, you can place the steps in a new sequence, create a Sequence Call step that calls the sequence, and loop on the Sequence Call step. Figure 5-15 shows the Loop Options tab on the Step Properties dialog box.

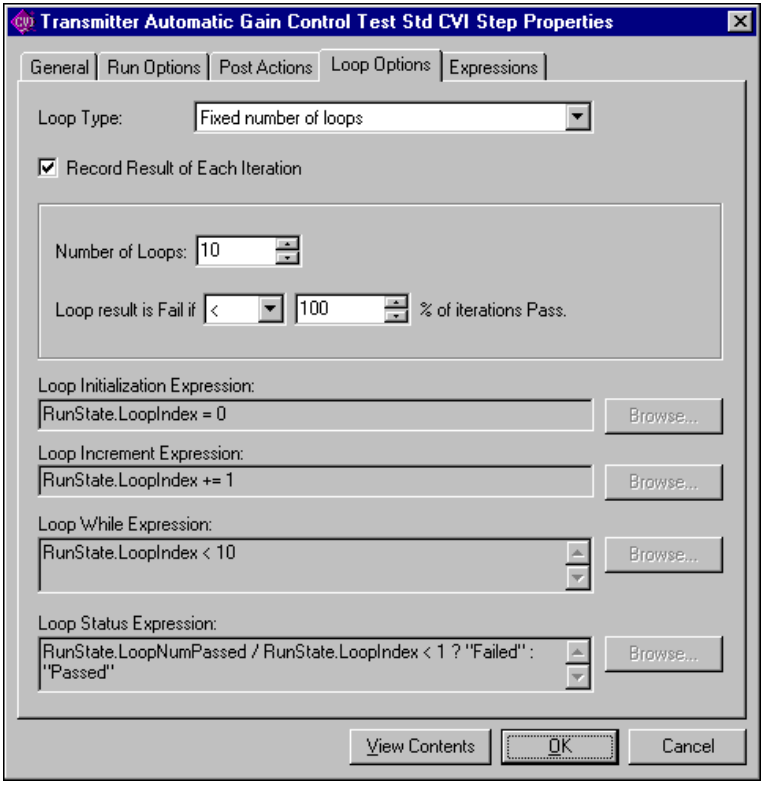

Figure 5-15. Loop Options Tab on the Step Properties Dialog Box

You can use the Loop Options tab on the Step Properties dialog box to specify the following options:

- **Loop Type**—Use this control to specify the type of looping for the step. The choices are as follows:
  - None—TestStand does not loop on the step. This is the default value.
  - Fixed number of loops—TestStand loops on the step a specific number of times and determines the final pass or fail status of the

step based on the percentage of loop iterations in which the step status is `Passed`.

– Pass/Fail count—TestStand loops on the step until the step passes or fails a specific number of times or until a maximum number of loop iterations complete. TestStand determines the final status of the step based on whether the specific number of passes or failures occur or the number of loop iterations reaches the maximum.

– Custom—This value allows you to customize the looping behavior for the step. You specify a Loop Initialization expression, a Loop Increment expression, a Loop While expression, and a final Loop Status expression. The following example code illustrates the order in which TestStand uses the loop expressions.

```
Loop_Initialization_Expression;
while (Loop_While_Expression == True)
    {
    Execute_Step;
    Loop_Increment_Expression;
    }
Loop_Status_Expression;
```

- **Record Result of Each Iteration**—If you enable this option, TestStand adds the step result to the sequence results list after each loop iteration. TestStand also adds the final result that it computes for the step loop as a whole if the Record Results property for the step is enabled. Refer to the *Result Collection* section in Chapter 6, *Sequence Execution*, for more information on result collection.

Note ***You do not have to use the Loop Options tab to cause execution to iterate or a step or series of steps. Instead, you can use a Goto step to create a loop inside your sequence. You can use the preconditions for the Goto step in combination with any number of variables to control the loop.***

You can use the Expressions tab to specify optional expressions that TestStand evaluates before or after it calls the step module.

Figure 5-16 shows the Expressions tab on the Step Properties dialog box.

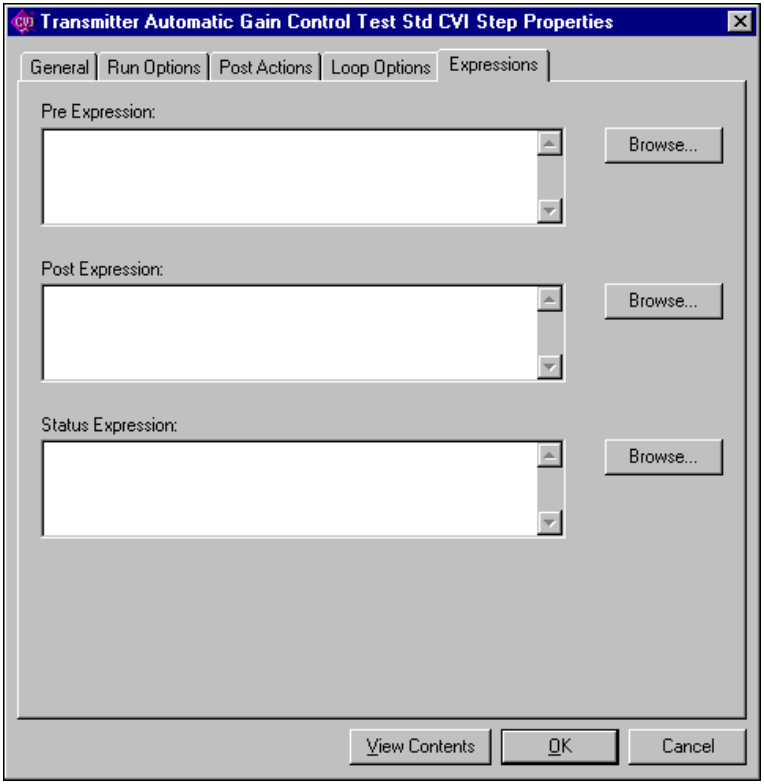

Figure 5-16. Expressions Tab on the Step Properties Dialog Box

The Expressions tab contains the following controls:

- **Pre Expression**—You can use this control to specify an expression that TestStand evaluates before it calls the step module. Usually, you use this expression to set the value of a custom step property from the values of other variables and properties.
- **Post Expression**—You can use this control to specify an expression that TestStand evaluates after it calls the step module. Usually, you use this expression to set the value of one of the subproperties in the `Result` property of the step from the values of other variables and properties.
- **Status Expression**—You can use this expression to set the status property for the step. Because the status is a string property, this expression must evaluate to a string.

If an expression is left empty, TestStand does not evaluate it.

Note ***Certain types of steps such as Numeric Limit Tests, String Value Tests, Pass/Fail Tests, and Statement steps reserve one or more of these expressions to perform operations specific to the type of step. In these cases, you cannot use the expressions that the step type reserves. The expressions appear dim in the tab.***

Parameters Tab

Sequences can have steps that call other sequences. A sequence can have parameters so that you can pass values to it and receive values from it. You define the parameters for a sequence in the Parameters tab.

Figure 5-17 shows the Parameters tab for an example sequence.

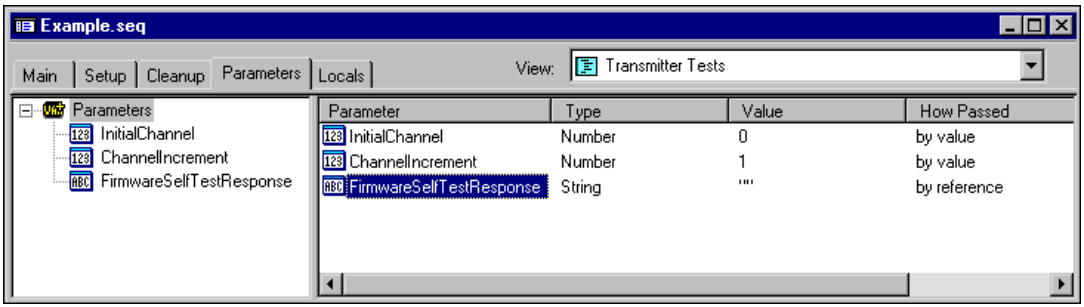

Figure 5-17. Parameters Tab

Parameters Tab Context Menu

You can display a context menu by right-clicking on the tree view or list view in the Parameters tab. The items in the context menu vary depending on the whether you right-click on a parameter, a parameter subproperty, or on the background area of the tree view, or on the background of the list view. The context menu can contain the following items.

Insert Parameter

The **Insert Parameter** menu item has a submenu in which you select the data type for the parameter you want to insert. Figure 5-18 shows the **Insert Parameter** submenu.

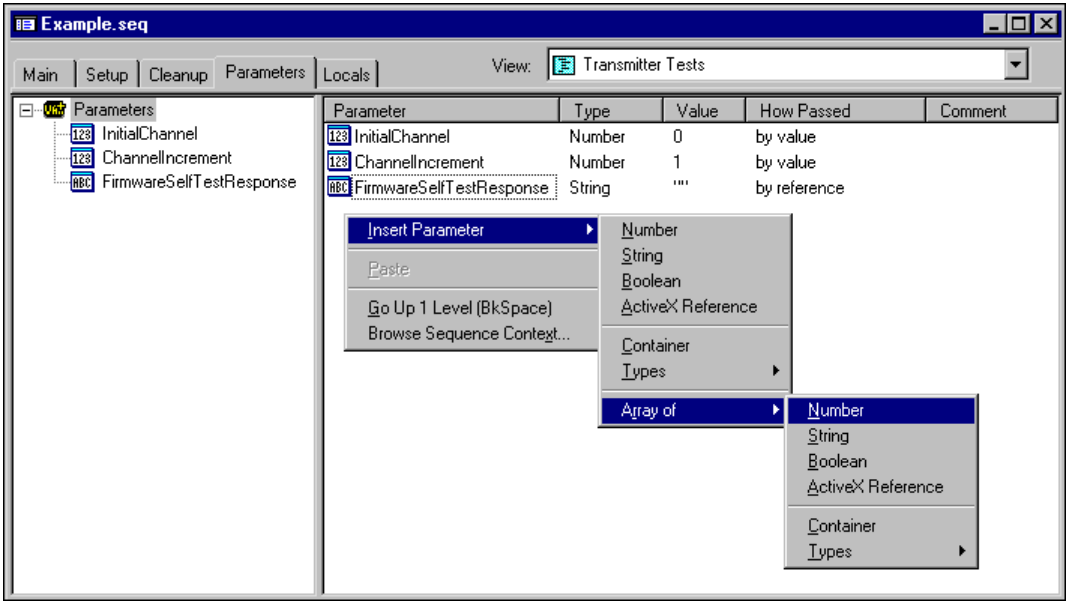

Figure 5-18. Insert Parameter Submenu

If you want to insert a parameter with a custom data type, you must create a named data type. You can create a named data type in the Sequence File Types view of the Sequence File window or in the Types Palette window. Refer to Chapter 9, *Types*, for more information on types and type editing. After you create the named data type, it appears in the **Types** submenu of the **Insert Parameter** submenu.

View Contents

The **View Contents** command selects the tree view node that corresponds to the currently selected item in the list view. The list view then displays the contents of the item. If the tree view is currently closed, it opens to show the selected node. You can use this command to view the subproperties of sequence parameters.

Go Up One Level

The **Go Up One Level** command selects the next higher level node in the tree view. The list view displays the contents of the newly selected node. If you invoke this command when the highest-level node is selected in the tree view, the Sequence File window displays the All Sequences View.

Browse Sequence Context

The **Browse Sequence Context** command displays a tree view that contains the names of variables and sequence parameters you can access from expressions and step modules when the sequence is running. This command also appears in the **View** menu of the sequence editor menu bar. Refer to the *View Menu* section in Chapter 4, *Sequence Editor Menu Bar*, for more information.

Rename

The **Rename** command allows you to edit the name of the selected parameter or subproperty.

Pass By Reference

The **Pass By Reference** command tells TestStand that the parameter is a reference to the argument that the calling sequence passes to the parameter. Passing a parameter by reference allows the subsequence to change the actual value of the argument in the calling sequence.

If you disable the **Pass By Reference** command for a parameter, TestStand copies the argument value that the calling sequence passes as the parameter. This prevents the subsequence from changing the value of the argument in the calling sequence. On the other hand, copying a large object or array that you pass as a parameter can degrade performance.

You enable the **Pass By Reference** command if you want to return a value from a subsequence to the calling sequence. You also can enable the option to reduce the time it takes to pass a large object or array to a subsequence. You disable the option if you want to guarantee that any changes that a subsequence makes to a parameter does affect the argument in the calling sequence.

Check Type

The **Check Type** option tells TestStand to verify that the data type of the argument you pass as a parameter is compatible with the data type of the parameter. For example, TestStand reports a run-time error if you set this option for a String parameter and then pass a numeric value instead.

Although type checking is usually a fast operation, you can turn this option off if you want to avoid any possible overhead. You also can turn this option off if you want to pass arguments with different types in the same parameter field for calls. Usually, you do this by specifying `Container` as the data type for the parameter and disabling this option. You can use the `PropertyExists` expression function to determine if the argument that a calling sequence passes to your container parameter contains a particular subproperty.

Parameter Properties

The **Properties** command displays a dialog box that you can use to change the default value for a parameter or one of its subproperties. TestStand uses the default values for all the parameters to a sequence when you run the sequence directly. When you call the sequence from a step in another sequence and the step passes fewer arguments than the sequence has, TestStand uses the default values for the remaining sequence parameters.

Locals Tab

Sequences can have any number of local variables. You can use local variables to hold values that you set or get in step modules. You also can use local variables for maintaining counts, for holding intermediate values, or for any other purpose. Refer to the *Using Data Types* section in Chapter 9, *Types*, for more information on using local variables.

Figure 5-19 shows the Locals tab for an example sequence.

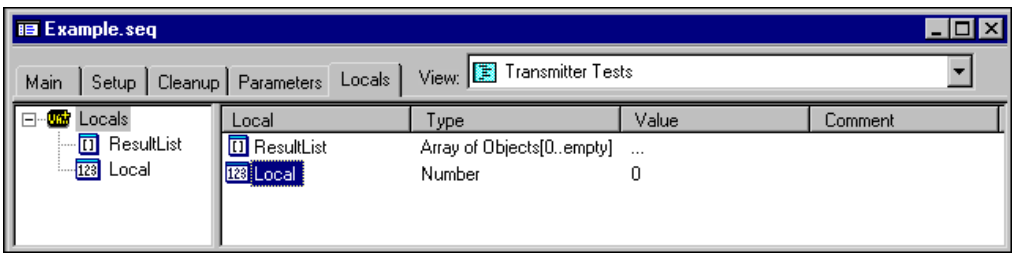

Figure 5-19. Locals Tab

Locals Tab Context Menu

You can display a context menu by right-clicking on the tree view or list view in the Locals tab. The items in the context menu vary depending on whether you right-click on a local variable, on a subproperty of a local variable, on the background area of the tree view, or on the background of the list view. The context menu can contain the following items.

Insert Local

The **Insert Local** menu item has a submenu in which you select the data type for the local variable you want to insert. Figure 5-20 shows the **Insert Local** submenu.

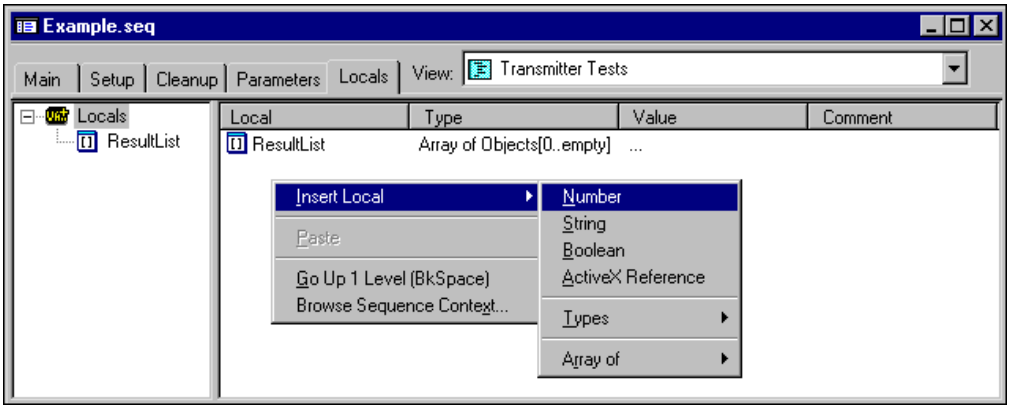

Figure 5-20. Insert Local Submenu

If you want to insert a local variable with a custom data type, you must create a named data type. You can create a named data type in the Sequence File Types view of the Sequence File window or in the Types Palette window. Refer to Chapter 9, *Types*, for more information on types and type editing. After you create the named data type, it appears in the **Types** submenu of the **Insert Local** submenu.

If you create an array, an Array Bounds dialog box appears. Refer to the *Specifying Array Sizes* section in Chapter 9, *Types*, for more information on the Array Bounds dialog box.

Notice that sequences always start with one local variable, `ResultList`. If you delete this local variable, TestStand cannot collect results for the sequence. Refer to the *Result Collection* section in Chapter 6, *Sequence Execution*, for more information on the results list.

View Contents

The **View Contents** command selects the tree view node that corresponds to the currently selected item in the list view. The list view then displays the contents of the item. If the tree view is currently closed, it opens to show the selected node. You can use this command to view the subproperties of local variables.

Go Up One Level

The **Go Up One Level** command selects the next higher level node in the tree view. The list view displays the contents of the newly selected node. If you invoke this command when the highest level node is selected in the tree view, the Sequence File window displays the All Sequences View.

Browse Sequence Context

The **Browse Sequence Context** command displays a tree view that contains the names of global variables, local variables, and sequence parameters you can access from expressions and step modules when the selected step is running. This command also appears in the **View** menu of the sequence editor menu bar. Refer to the *View Menu* section in Chapter 4, *Sequence Editor Menu Bar*, for more information.

Rename

The **Rename** command allows you to edit the name of the selected local variable or subproperty.

Properties

The **Properties** command displays a dialog box you can use to change the default value for the selected local variable or subproperty. TestStand sets the values of the local variables to their default values when the sequence begins executing. If the local variable or subproperty is an array, you can use the Bounds tab on the dialog box to change the array bounds.

Preconditions Dialog Box

TestStand has several features that you can use to control the flow of execution in a sequence. These include the post actions for a step, the preconditions for a step, and the Goto step type. You can combine these features in various ways. For example, you can use the preconditions on a Goto step to specify when to loop back to an earlier statement. This section discusses the Preconditions dialog box.

You can display the Preconditions dialog box by clicking on the **Preconditions** button on the Sequence Properties dialog box or by clicking on the **Preconditions** button on the Step Properties dialog box.

Figure 5-21 shows the Preconditions dialog box for a sequence.

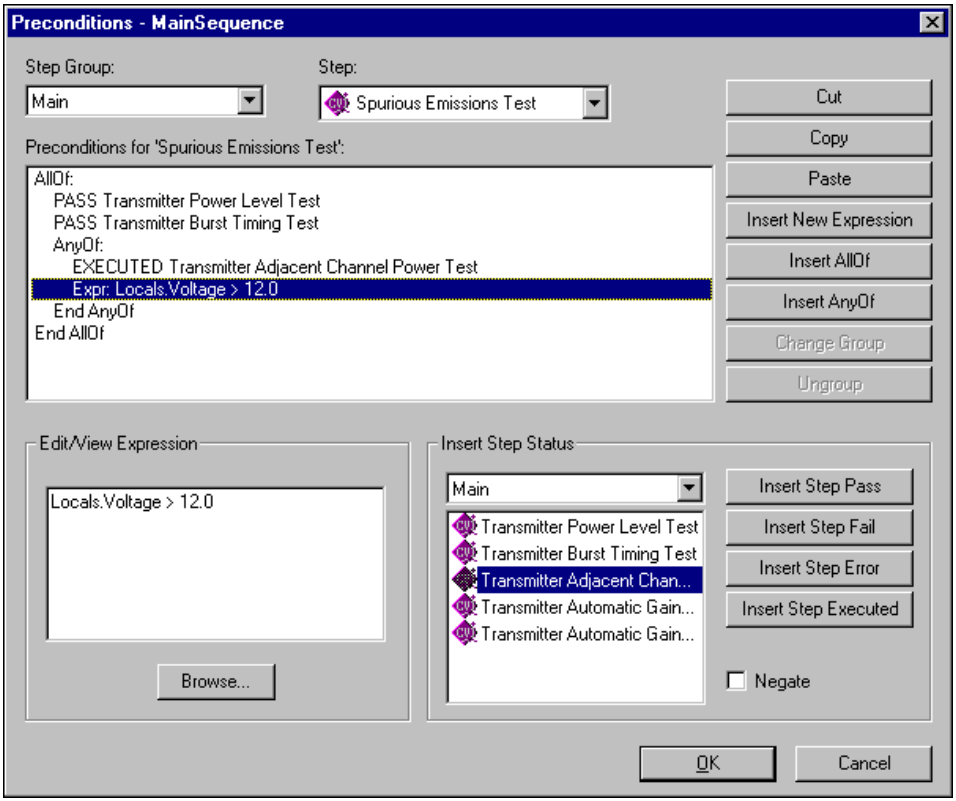

Figure 5-21. Preconditions Dialog Box for a Sequence

The Step Group and Step controls indicate the step to which the preconditions apply. When you invoke the dialog box from the Sequence Properties dialog box, you can use these controls to select any step group and step in the sequence. When you invoke the dialog box from the Step Properties dialog box, these controls are not operable.

The Preconditions list box shows the preconditions of the step. The label above the list box includes the name of the step. The following items can appear in the Preconditions list box:

- **Arbitrary expression**—The list box marks lines that contain arbitrary expressions by beginning them with the `Expr:` tag.
- **Step status expression**—Step status expressions refer to the status of other steps in the sequence. In the list box, step status expressions begin with `PASS`, `NOT PASS`, `FAIL`, `NOT FAIL`, `ERROR`, `NOT ERROR`, `EXECUTED`, or `NOT EXECUTED`, followed by the name of the step.
- **AllOf block**—An AllOf block brackets multiple expressions and evaluates to `True` only if all the expressions in the block evaluate to `True`. Each AllOf block consists of a line containing `AllOf` and another line containing `End AllOf`.
- **AnyOf block**—An AnyOf block brackets multiple expression and evaluates to `True` if one or more expressions in the block evaluate to `True`. Each AnyOf block consists of a line containing `AnyOf` and another line containing `End AnyOf`.

You can nest one or more blocks within another block. A block treats a nested block as just another expression.

The following buttons appear in the Preconditions dialog box:

- **Cut** or **Copy—**If you use these buttons on an `AllOf` or `AnyOf` line in the list box, it cuts or copies the entire block. The **Cut** and **Copy** buttons are dim when an `End AllOf` or `End AnyOf` line is currently selected.
- **Insert New Expression**—Use this button to insert an empty arbitrary expression below the current line in the Preconditions list box.
- **Insert AllOf**—Use this button to insert an empty AllOf block below the current line in the Preconditions list box.
- **Insert AnyOf**—Use this button to insert an empty AnyOf block below the current line in the Preconditions list box.

To nest a block within an existing block, select the `AllOf` or `AnyOf` line of the existing block and click on the **Insert AnyOf** or **Insert AllOf** button.

To add a block at the same level as an existing block, select the `End AllOf` or `End AnyOf` line of the existing block and click on the **Insert AnyOf** or **Insert AllOf** button.

When you select an `AllOf` line, you can use the **Change to AnyOf** button to change the block to an AnyOf block. When you select an `AnyOf`, you can use the **Change to AllOf** button to change the block to an AllOf block.

When you select the `AllOf` line or `AnyOf` line of a block that contains only one expression or that is nested within another block, you can use the **Ungroup** button to remove the block but keep its contents.

You use the Edit/View Expression text box to view or modify an expression line in the Preconditions list box. You can enter or modify the expression manually in the text box. You also can use the **Browse** button to display an expression browser dialog box in which you can interactively build an expression from lists of available variables, properties, and expression operators. Refer to Chapter 8, *Sequence Context and Expressions*, for more information on expressions.

You use the Insert Step Status section of the dialog box to design a step status expression. You use the ring control and list box to choose a step group and step in the sequence. You can use the Negate checkbox to negate the meaning of an expression. The following describes the command buttons in this section:

- **Insert Step Pass**—Inserts an expression that is `True` if the status for the most recent execution of the selected step is `Passed`.
- **Insert Step Fail**—Inserts an expression that is `True` if the status for the most recent execution of the selected step is `Failed`.
- **Insert Step Error**—Inserts an expression that is `True` if the status of the most recent execution of the selected step is `Error`, which indicates that a run-time error occurred in the step.
- **Insert Step Executed**—Inserts an expression that is `True` if the status for the most recent execution of the selected step is anything other than an empty string.

If, for example, you select the ROM step, enable the Negate checkbox, and click on the **Insert Step Pass** button, TestStand inserts a line containing `NOT PASS ROM` in the list box.

Sequence File Globals View

Each sequence file can contain any number of global variables. Figure 5-22 shows the contents of the Sequence File Globals view for an example sequence.

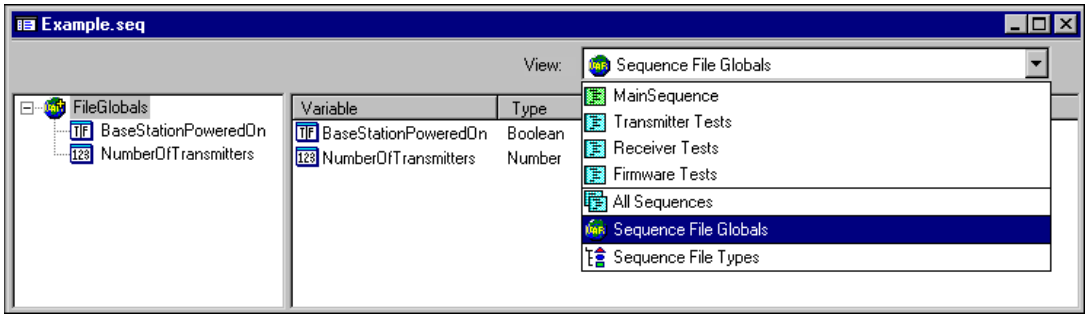

Figure 5-22. Sequence File Globals View for an Example Sequence

Lifetime and Scope of Sequence File Global Variables

Each time you begin a new execution or TestStand loads a sequence file dynamically during execution, TestStand creates a separate run-time copy of the global variables and initializes them to their default values. If TestStand unloads the sequence file during execution, TestStand destroys the global variables. If TestStand reloads the sequence file later during the same execution, TestStand creates a new copy of the global variables and initializes them to their default values.

Any sequence in the file can access the global variables for the file. A subsequence can access the global variables in the sequence file that contains the calling sequence. It also can access the global variables in the process model file and the client sequence file explicitly. A subsequence can do this in an expression or in a call to the TestStand ActiveX API in a code module. The sequence context contains references to the calling sequence, the main sequence in the client sequence file, and the process model entry point sequence. Refer to Chapter 8, *Sequence Context and Expressions*, for more information.

Sequence File Globals View Context Menu

You can display a context menu by right-clicking on the tree view or list view in the Globals View. The items in the context menu vary depending on the whether you right-click on a global variable on a subproperty of a global variable, on the background area of the tree view, or on the background of the list view. The context menu can contain the following items.

Insert Global

The **Insert Global** menu item has a submenu in which you select the data type for the sequence file global variable you want to insert. Figure 5-23 shows the **Insert Global** submenu.

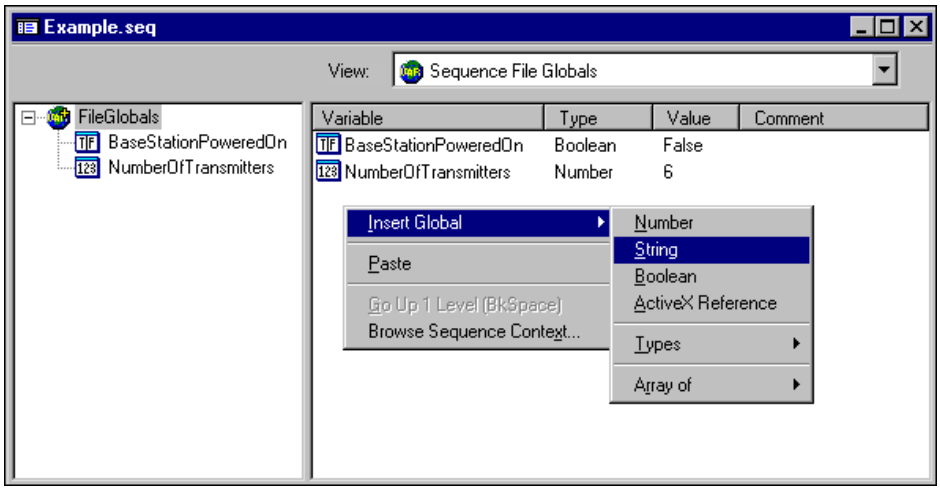

Figure 5-23. Insert Global Submenu

If you want to insert a global variable with a custom data type, you must create a named data type. You can create a named data type in the Sequence File Types view of the Sequence File window or in the Types Palette window. Refer to Chapter 9, *Types*, for more information on types and type editing. After you create the named data type, it appears in the **Types** submenu of the **Insert Global** submenu.

If you create an array, an Array Bounds dialog box appears. Refer to the *Specifying Array Sizes* section in Chapter 9, *Types*, for more information on the Array Bounds dialog box.

View Contents

The **View Contents** command selects the tree view node that corresponds to the currently selected item in the list view. The list view then displays the contents of the item. If the tree view is currently closed, it opens to show the selected node. You can use this command to view the subproperties of sequence file global variables.

Go Up One Level

The **Go Up One Level** command selects the next higher level node in the tree view. The list view displays the contents of the newly selected node.

Browse Sequence Context

The **Browse Sequence Context** command displays a tree view that contains the names of the station global variables and sequence file global variables you can access from expressions and step modules when sequences in the file are running. This command also appears in the **View** menu of the sequence editor menu bar. Refer to the *View Menu* section in Chapter 4, *Sequence Editor Menu Bar*, for more information.

Rename

The **Rename** command allows you to edit the name of the selected global variable or subproperty.

Properties

The **Properties** command displays a dialog box you can use to change the default value for the selected global variable or subproperty. TestStand sets the values of the global variables to their default values when the sequence begins executing. If the globals variable or subproperty is an array, you can use the Bounds tab on the dialog box to change the array bounds.

Sequence File Types View

Sequence files contain the type definitions for every step, property, and variable that the file contains. You can view the types that a sequence file contains by selecting Sequence File Types from the sequence file View ring.

Figure 5-24 shows the Sequence File Type view for an example sequence file.

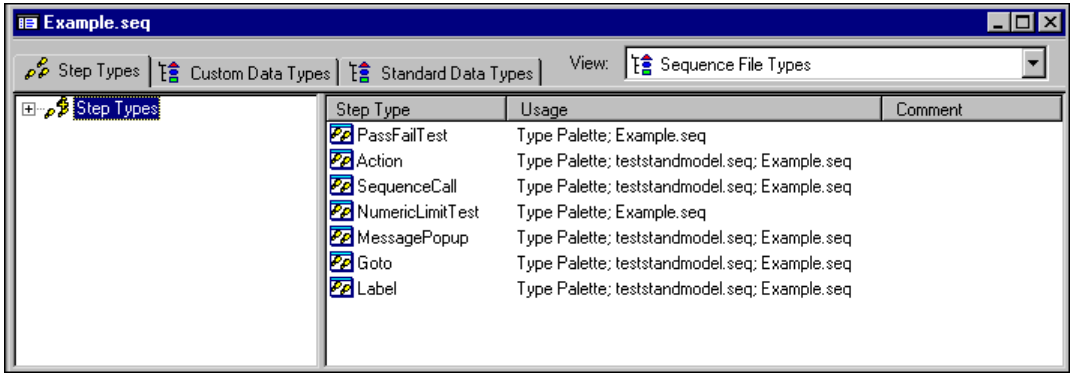

Figure 5-24. Step Types Tab in Sequence File Types View

Refer to Chapter 9, *Types*, for more information on the types and type editing.

6

Sequence Execution

This chapter describes the execution of sequences in TestStand. It also describes the Execution window in the TestStand sequence editor.

Sequence Editor and Run-Time Operator Interfaces

TestStand ships with a fully functional sequence editor and run-time operator interfaces. Like the sequence editor, the run-time operator interfaces allow you start multiple concurrent executions, set breakpoints, and perform single-step debugging. However, the *run-time operator interfaces* do not display sequence variables, sequence parameters, and step properties, or allow you to use watch expressions.

What is an Execution?

An *execution* is an object that TestStand creates to contain all the information that TestStand uses to run your sequence and the subsequences it calls. 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. Each execution runs in a different thread.

Whenever TestStand begins executing a sequence, it makes a *run-time copy* of the sequence local variables and the custom properties of the sequence steps. If the sequence calls itself recursively, TestStand creates a separate run-time copy of the local variables and custom step properties for each activation 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.

For each active execution, 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 on the Station Options dialog box provides a number of execution options that control tracing, breakpoints, and result collection.

Refer to the *Station Options* section in Chapter 4, *Sequence Editor Menu Bar*, for more information on the station execution options.

Starting an Execution

You can initiate an execution by launching a sequence through a model entry point, by launching a sequence directly, or by executing a group of steps interactively.

Execution Entry Points

You can start execution through an entry point only if the active window is for a sequence file that contains a sequence with the name `MainSequence`. A list of entry points appears in the **Execute** menu of the sequence editor.

Each 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 entry point from the **Execute** menu, you are actually running an entry point sequence in a process model file. The entry point sequence, in turn, invokes the main sequence one or more times.

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

Refer to the *Process Models* section in Chapter 1, *TestStand Architecture Overview*, and to Chapter 13, *Process Models*, for more information on process models.

Executing a Sequence Directly

To execute a sequence without using a process model, select the **Run *Sequence Name*** item in the **Execute** menu, where ***Sequence Name*** is the name of the sequence you are currently viewing. This command executes the sequence directly, skipping the process model operations such as UUT identification and test report generation. You can execute any sequence this way, not just main sequences. Usually, you execute a sequence in this way to perform unit testing or debugging.

Interactively Executing Steps

You can execute selected steps in a sequence interactively by choosing **Run Selected Steps** or **Loop Selected Steps** from the context menu in the sequence editor or by clicking on the **Run Tests** or **Loop Tests** buttons in the run-time operator interfaces.

In interactive mode, only the selected steps in the sequence execute, regardless of any branching logic that the sequence contains. The selected steps run in the order in which they appear in the sequence. TestStand does not evaluate step preconditions, so TestStand runs every selected step. However, TestStand does honor the Run Mode property for a selected step, that is force fail, force pass or skip. In addition, TestStand does not perform code module preloading for selected steps in interactive mode.

If you execute steps in a Sequence File window, you initiate the interactive execution as an independent top-level execution, or *root interactive execution*. When you do so, you create a new execution. You can set station options to control whether the Setup and Cleanup step groups of the sequence run as part of the root interactive execution. Root interactive executions do not invoke the process model.

If you execute steps in an Execution window when the execution is suspended, you initiate the interactive execution as a *nested interactive execution*, which is an extension of the suspended execution. When you do this, the selected steps run within the context of the normal execution.

Sequence Editor Execution Window

Operator interfaces usually provide a separate Execution window or view for each execution you start. For instance, the sequence editor displays each execution in a separate window.

The Execution window is divided into several areas. The top half of the window contains the Steps tab, Context tab, and Report tab. The bottom half of the window is divided into the Call Stack pane and the Watch Expression pane. A status bar appears at the bottom edge of the window.

The Threads selection ring lists all the threads running in the execution. In the current version of TestStand, each execution can have only one thread. There is no limit on the number of simultaneous executions. Future versions of TestStand will allow multiple threads in each execution. Each entry in the selection ring contains the name of the active sequence in the call stack for the thread. When you select a different thread from the

selection ring, the contents of the various tabs and panes in the Execution window change to display the state of the new thread.

Steps Tab

The Steps tab displays a list of the steps in the step group that is currently executing. Figure 6-1 shows the Execution window Steps tab.

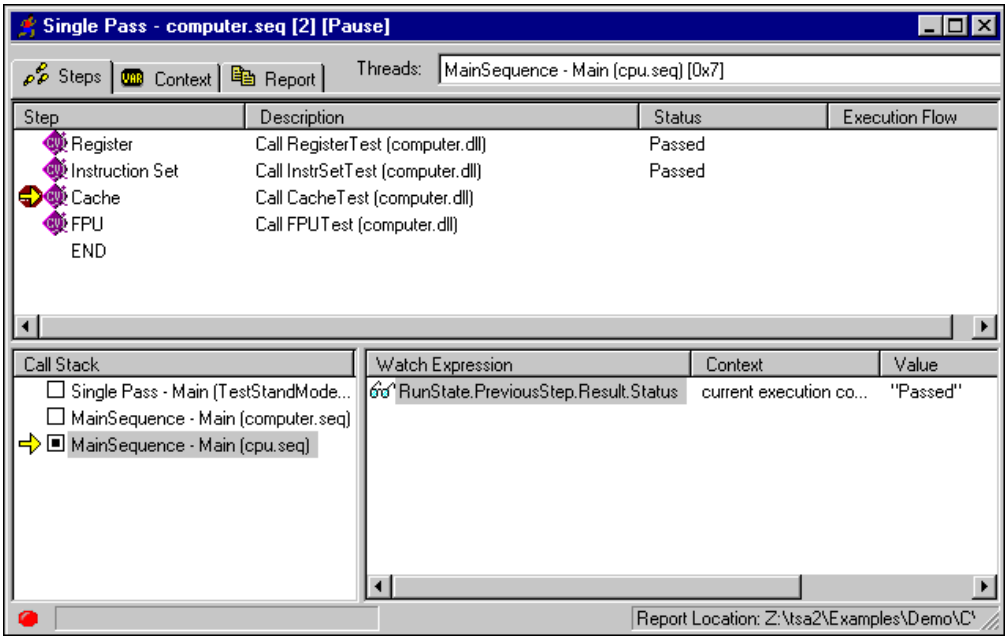

Figure 6-1. Steps Tab in the Sequence Editor Execution Window

When execution is suspended at a breakpoint, you can view the steps of any of the sequences that are active on the call stack. Use the Call Stack pane to select the active sequence to display in the Steps tab.

Tracing

If tracing is enabled, the sequence editor displays the progress of an execution by placing a yellow arrow icon to the left of the icon for the currently executing step in the Steps tab. The arrow icon is called the *execution pointer*. When execution suspends at a breakpoint, the Steps tab displays the execution pointer next to the step that will run when execution resumes. After each step completes, the Execution window updates the contents of the Steps tab, the position of the execution pointer, and the values of any watch expressions in the Watch panel.

If tracing is disabled, the Execution window does not update until execution suspends at a breakpoint. The Step tab might display no steps at all, or it might contain the steps and execution pointer that it displayed at the most recent breakpoint.

Usually, you disable tracing if you want to avoid using a lot of computer time to display the progress of your execution. You can use the **Tracing Enabled** item in the **Execute** menu to enable or disable tracing. You also can control tracing from the Execution tab in the Station Options dialog box.

Debugging

The sequence editor and operator interfaces allow you to set breakpoints, to step over or step into steps, to step out of sequences, and to set the next step to execute. You also can terminate execution, abort execution, toggle tracing, and run or loop on selected steps. In the sequence editor, these commands are in the **Execute** menu and the **Debug** menu. Refer to the *Execute Menu* and *Debug Menu* sections, in Chapter 4, *Sequence Editor Menu Bar*, for more information on debugging commands.

Steps Tab Columns

As shown in Figure 6-1, the Steps tab contains the following columns:

- **Step**—Displays the name and icon of the step. You can click to the left of the step icon to toggle the breakpoint for the step.
- **Description**—Displays a description of the step that varies according to the type of step and the module adapter that is uses.
- **Status**—Displays the value of the status property for the step. If the step has not yet executed, its status is an empty string. After the step executes, its status reflects the result of its execution. Possible status values can vary based on the type of step. Typical values include `Passed`, `Failed`, `Done`, and `Error`. Refer to the *Step Status* section, later in this chapter, for more information on step status values.
- **Execution Flow**—Indicates the properties that the step uses to control the flow of execution in the sequence. The values that can appear in this column and their meanings are as follows:
  - Pre—Indicates that the step has a precondition.
  - Post—Indicates that the step has a post action.
  - Loop—Indicates that step is configured to loop.
  - Skip—Indicates that the run mode of the step is Skip.

- Force Pass—Indicates that the run mode of the step is Force Pass.
- Force Fail—Indicates that the run mode of the step is Force Fail.

Steps Tab Context Menu

When execution is suspended at a breakpoint, you can display a context menu for the Steps tab by right-clicking on the name or icon of a step. The context menu can contain the following items.

Toggle Breakpoint

The **Toggle Breakpoint** command sets or clears the breakpoint state for the selected steps.

Run Mode

The **Run Mode** menu item displays a submenu from which you can set the run mode for the selected steps. The following run mode values are possible.

- **Force Pass**—TestStand does not execute the step and does not evaluate its preconditions. Instead, TestStand sets the status of the step to `Passed` automatically.
- **Force Fail**—TestStand does not execute the step and does not evaluate its preconditions. Instead, TestStand sets the status of the step to `Failed` automatically.
- **Skip**—TestStand does not execute the step and does not evaluate its preconditions. Instead, TestStand sets the status of the step to `Skipped` automatically.
- **Normal**—TestStand executes the step normally. This is the default value.

Set Next Step

The **Set Next Step** command tells TestStand to start from the selected step when you resume execution.

Run Selected Steps

The **Run Selected Steps** command runs the selected steps in interactive mode.

Loop Selected Steps

The **Loop Selected Steps** command loops on the selected steps in interactive mode. Before running the steps, this command displays a dialog box in which you specify the number of times to loop, and a stop condition that TestStand evaluates after it executes each step.

Show Step in Context Tab

The **Show Step in Context Tab** command switches from the Steps tab to the Context tab and positions the selection in the Context tab on the step that has the focus in the Steps tab. Usually, you use this command to view the values of the custom properties of a step after it executes.

Properties

The **Properties** command displays the Step Properties dialog box for the selected step. Usually, most controls on the dialog box are disabled, because you cannot edit most step properties during an execution.

Context Tab

The Context tab displays the sequence context for the sequence invocation that is currently selected in the Call Stack pane. The sequence context contains all the variables and properties that the steps in the selected sequence invocation can access.

You use the Context tab to examine and modify the values of these variables and properties. You can drag individual variables or properties from the Context tab to the Watch Expression pane so that you can view changes in their values while you single-step or trace through the sequence. When execution completes, the Context tab becomes hidden. Refer to Chapter 8, *Sequence Context and Expressions*, for more information on sequence contexts.

Figure 6-2 shows the Context tab for an example sequence in which execution is suspended.

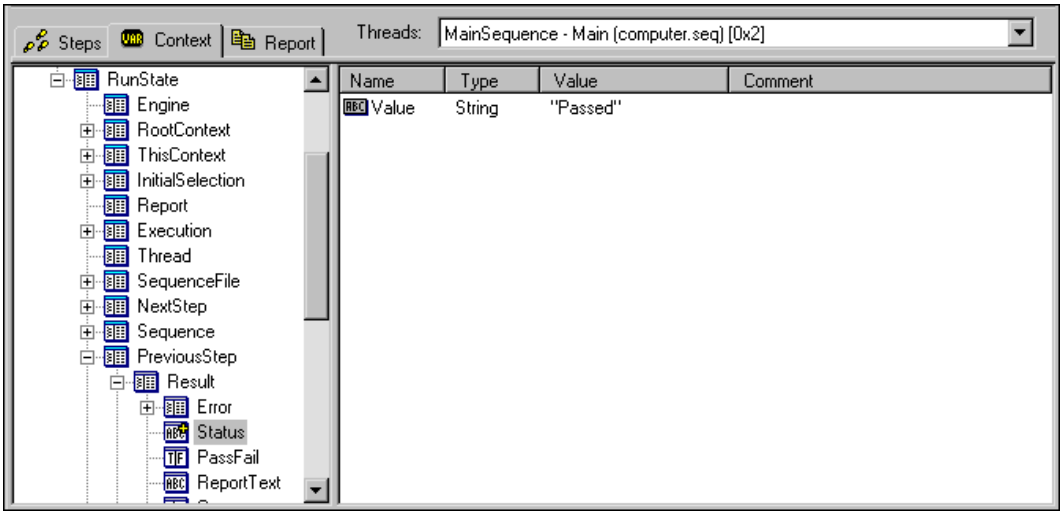

Figure 6-2. The Context Tab in an Execution Window

The sequence context contains entries for each step in the sequence. You can locate a particular step in the Context tab by selecting the **Show Step in Context Tab** item from the context menu on a step in the Steps tab.

Context Tab Context Menu

When execution is suspended, you can display a context menu for the Context tab by right-clicking on a variable or property. The context menu can contain the following items.

View Contents

The **View Contents** command selects the tree view node that corresponds to the currently selected item in the list view. The list view then displays the contents of the item. You use this command on variables or properties to view their subproperties.

Refresh

When an execution suspends, other executions that are not suspended can change the values of station global variables that appear in the Context tab for the suspended execution. You can use the **Refresh** command to update the Context tab so that it displays the current values of the station global variables.

Object Properties

The **Properties** command displays a dialog box you can use to change the current value of the selected variable or property.

Report Tab

The Report tab displays the report for the current execution. Usually, the Report tab is empty until execution completes.

By default, an execution generates a report only when you start the execution through a model entry point such as `Test UUTs` or `Single Pass`. You can set options that control report generation by selecting the **Report Options** item in the **Configure** menu. The default process model can generate reports in either HTML or ASCII text formats.

The Report tab in the sequence editor can display reports in HTML, ASCII text, or Rich Text Format (RTF). You also can use an external application to view reports in these or other formats by selecting the **Launch Report Viewer** command in the **View** menu when an Execution window is active. You can use the **External Viewers** menu item in the **Configure** menu to specify the external application that TestStand launches to display a particular report format.

The sequence editor uses the Internet Explorer component to display HTML reports. If your sequence generates a very large number of results, it can take a substantial amount of time for this component to load and display the report. If the report does not appear in an acceptable amount of time after the process model generates it, you can use the **Report Options** item in the **Configure** menu to specify a filter expression that reduces the number of results in the report. Another way to display a large report quickly is to change the report format to ASCII text.

When you select the Report tab, TestStand hides the Call Stack and Watch Expression panes so that it can use the entire Execution window to display the report. Refer to Chapter 13, *Process Models*, for more information on report generation.

Figure 6-3 shows an HTML report for an example sequence.

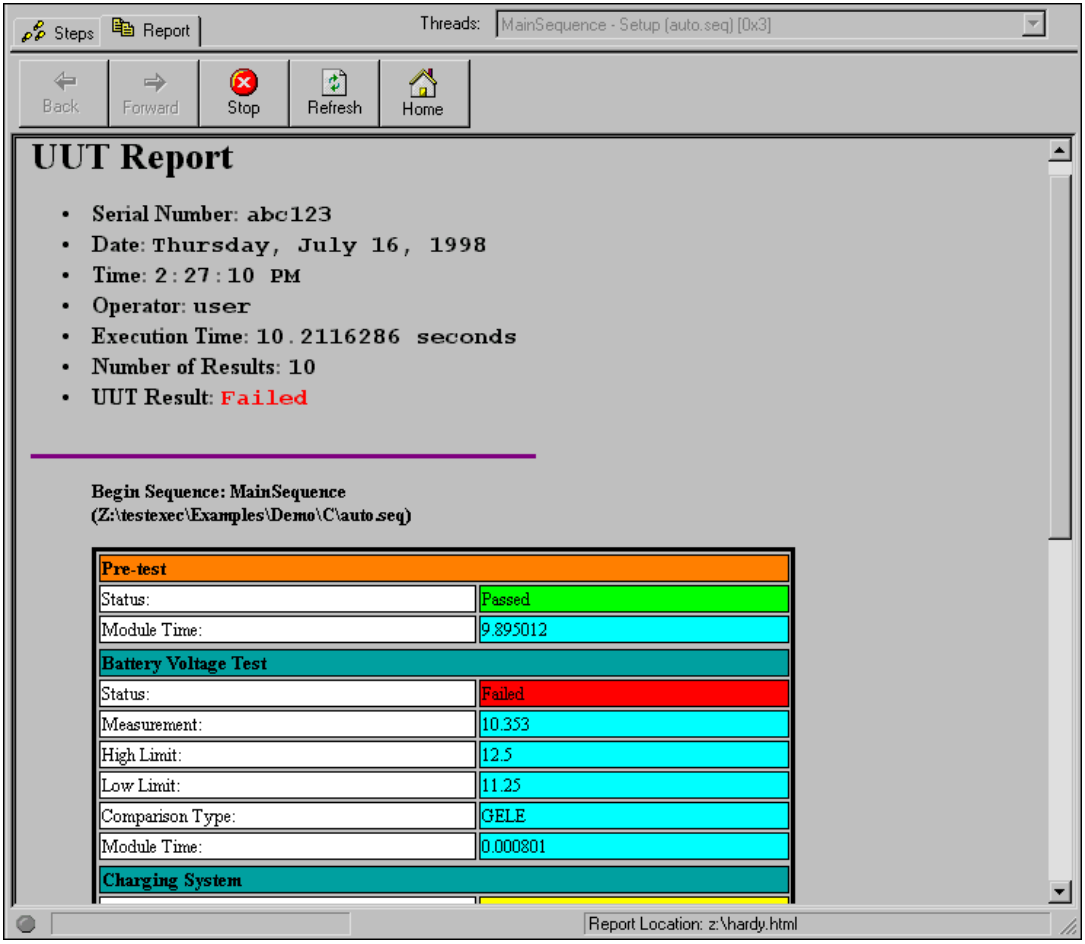

Figure 6-3. HTML Report for an Example Sequence

Call Stack Pane

Usually, when a step invokes a subsequence, the sequence that contains the calling step waits for the subsequence to return. The subsequence invocation is nested in the invocation of the calling sequence. The sequence that is currently executing is the *most nested sequence*. The chain of active sequences that are waiting for nested subsequences to complete is called the call stack. The last item in the call stack is the most nested sequence invocation.

The Call Stack pane displays the call stack for the execution thread that is currently selected in the Thread selection ring. A yellow pointer icon appears to the left of the most nested sequence invocation. The call stack in Figure 6-4 shows that the `Test UUTs` model entry is calling the main sequence in `Computer.seq`, which in turn is calling the main sequence in the `CPU.seq`.

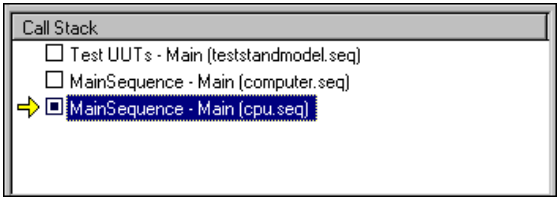

Figure 6-4. Call Stack Pane while Suspended in a Subsequence

When execution suspends, you can select a sequence invocation in the call stack by clicking on its radio button. The Steps tab displays the steps for the sequence invocation. The Watch Expression pane evaluates its watch expressions using the sequence context for the selected sequence invocation. In Figure 6-5, the main sequence from `Computer.seq` is selected in the Call Stack pane.

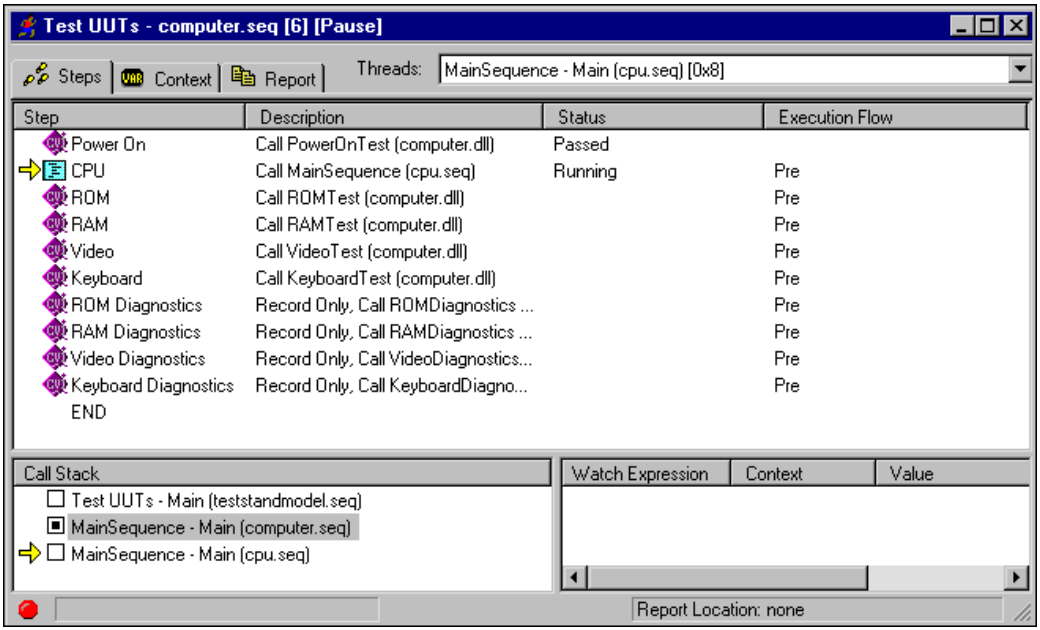

Figure 6-5. Steps Tab Displaying a Sequence Invocation in the Middle of the Call Stack

When the steps view displays the steps for a call stack item that is not the most nested item, a green pointer icon appears next to the sequence call step that is waiting to complete.

Watch Expression Pane

The Watch Expression pane displays the values of watch expressions that you enter. TestStand updates the values in the Watch Expression pane when execution suspends at a breakpoint. If tracing is enabled, TestStand also updates the values after executing each step.

Usually, you enter watch expressions to monitor the values of variables and properties as you trace or single-step through a sequence. You can drag individual variables or properties from the Context tab to the Watch Expression pane.

Figure 6-6 shows several example watch expressions in the Watch Expression pane.

Watch Expression	Value	Type
Locals.ProcessorSpeed	400	Number
Locals.NumSlots	6	Number
Locals.FreeDiskSpaceInGigs	8.4	Number
Locals.FreeDiskSpaceInGigs > 6.0	True	Boolean
RunState.PreviousStepIndex	2	Number
RunState.PreviousStep.Result.Status	"Passed"	String
RunState.PreviousStep.Result.PassFail	True	Boolean

Figure 6-6. Watch Expression Pane

When execution is suspended, you can display a context menu by right-clicking in the Watch Expression pane. The items in the context menu vary depending on the whether you right-click on a watch expression or its icon, or on the background of the pane. The context menu can contain the following items.

Edit Expression

The **Edit Expression** command displays an expression browser dialog box in which you can edit the selected watch expression.

Add Watch

The **Add Watch** command inserts an empty watch expression into the pane and then displays an expression browser dialog box in which you can edit the new expression.

Modify Value

The **Modify Value** command displays a dialog box in which you can edit the value of the selected watch expression. TestStand dims the **Modify Value** command if the selected expression does not evaluate to a single variable or property value. For example, you can modify the value of `Locals.X` but not the value of `Locals.X + 5`.

Refresh

When an execution suspends at a breakpoint, other executions that are not suspended can change the values of station global variables that a watch expression refers to. You can use the **Refresh** command to update the Watch Expression pane so that it displays the current values for watch expressions that contain station global variables.

Status Bar

Figure 6-7 shows the status bar for an Execution window in the sequence editor.

Figure 6-7. Execution Window Status Bar

The status bar contains the following four elements arranged from left to right.

- **Execution Status LED**—The LED is green while the execution runs, red when the execution suspends, and dark gray when the execution completes.
- **Progress Indicator Bar**—Through the TestStand ActiveX API, a step module can request that an operator interface program display an indication of the step's progress toward completion. Usually, a step module developer uses this feature if the step takes longer than a few seconds to complete. The Execution window displays the degree of progress in the Progress Indicator bar. The default process model also uses the Progress Indicator bar to display progress while it generates the test report.

- **Status Message**—Through the TestStand ActiveX API, step modules can request that an operator interface program display a short message. The Execution window displays these messages to the right of the Progress Indicator bar.
- **Report Location**—Each execution has its own test report. The Execution window displays the location of the test report for the execution in the rightmost box on the status bar. Usually, the process model fills in the Report Location with the pathname of the file to which the model writes the report.

Result Collection

TestStand can automatically collect the results of each step. You can configure this for each step in the Run Options tab of the Step Properties dialog box. You can disable result collection for an entire sequence in the Sequence Properties dialog box. You can completely disable result collection on your computer in the Station Options dialog box.

Each sequence has a `ResultList` local variable that is initially an empty array of container properties. TestStand appends a new container property to the end of the `ResultList` array before a step executes. This container property is called the *step result*. After the step executes, TestStand automatically copies the contents of the `Result` subproperty for the step into the step result.

Each step type can define different contents for its `Result` subproperty. 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 its step result, it also adds information such as the name of the step and its position in the sequence. For a step that calls a subsequence, TestStand also adds the `ResultList` array variable from the subsequence.

Figure 6-8 shows the result of a Numeric Limit Test step in expanded form on the Execution window Context tab.

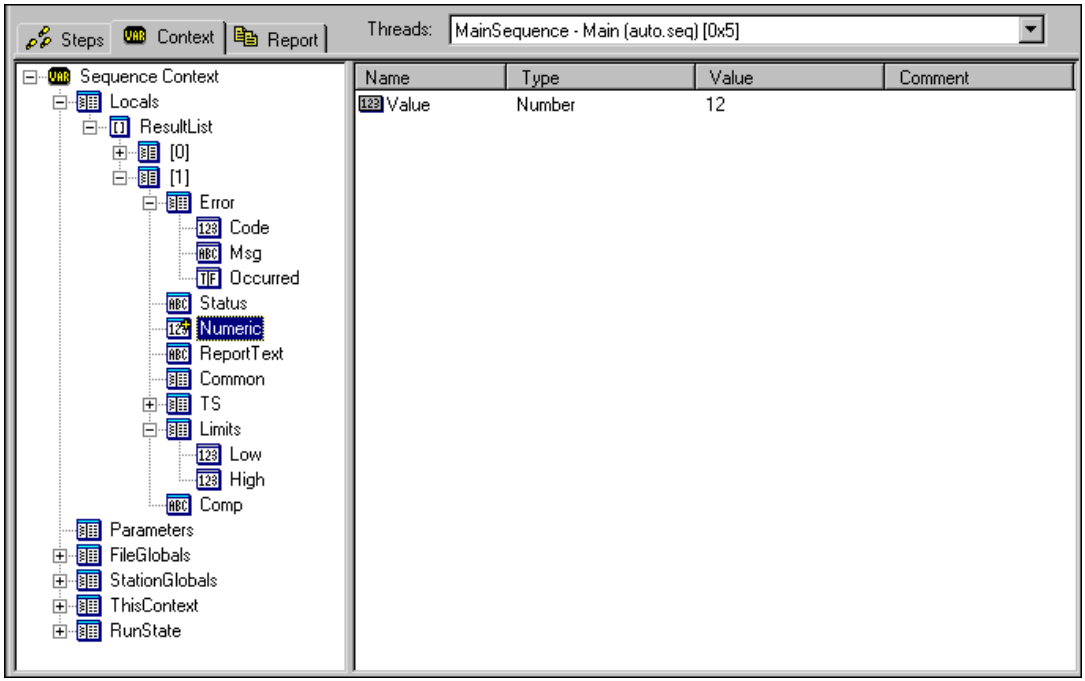

Figure 6-8. A Result in a ResultList Array

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

Custom Result Properties

Because each step type can have a different set of subproperties under its `Result` property, the step result varies according to the step type. Table 6-1 lists the custom properties that the step result can contain for steps that use one of the built-in step types.

Table 6-1. Custom Properties in the Step Results for Steps That Use the Built-In Step Types

Custom Step Property	Step Types that Use the Property
`Error.Code`	All
`Error.Msg`	All
`Error.Occurred`	All
`Status`	All
`Common`	All
`Numeric`	`NumericLimitTest`
`PassFail`	`PassFailTest`
`String`	`StringLimitTest`
`ButtonHit`	`MessagePopup`
`Response`	`MessagePopup`
`ExitCode`	`CallExecutable`
`NumLimitsInFile`	`LimitLoader`
`NumRowsInFile`	`LimitLoader`
`NumLimitsApplied`	`LimitLoader`
`ReportText`	All
`Limits.Low`	`NumericLimitTest`
`Limits.High`	`NumericLimitTest`
`Comp`	`NumericLimitTest`

The `Common` result subproperty uses the `CommonResults` custom data type. The `Common` property is a subproperty of the `Result` property for every built-in 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` type.

The `Limits.Low`, `Limits.High`, and `Comp` properties are not subproperties of the `Result` property. Thus, TestStand does not include them in the step results automatically. Depending on options you set, the default process model uses the TestStand ActiveX API to include these properties in the step results for steps that contain them.

Standard Result Properties

In addition to copying custom step properties, TestStand also adds a set of standard properties to each step result. TestStand adds standard result properties to the step result as subproperties of the `TS` property. Table 6-2 lists the standard result properties.

Table 6-2. Standard Step Result Properties

Standard Result Property	Description
`TS.StartTime`	Time at which the step began executing. The time is in terms of the number of seconds since the TestStand engine initialized.
`TS.TotalTime`	Number of seconds the step took to execute. This time includes the time for all step options including preconditions, expressions, post actions, module loading, and module execution.
`TS.ModuleTime`	Number of seconds the step module took to execute.
`TS.Index`	Zero-based position of the step in the step group.
`TS.StepName`	Name of the step.
`TS.StepGroup`	Step group that contains the step. The value is `Main`, `Setup`, or `Cleanup`
`TS.Id`	A number that TestStand assigns to the step result. The number is unique with respect to all other step results in the current TestStand session.

Table 6-2. Standard Step Result Properties (Continued)

Standard Result Property	Description
`TS.InteractiveExeNum`	A number that TestStand assigns to an interactive execution. The number is unique with respect to all other interactive executions in the current TestStand session. TestStand adds this property only if you run the step interactively.
`TS.StepType`	Name of the step type.

Subsequence Results

If a step calls a subsequence or generates a call to a callback sequence, TestStand creates a special step result subproperty to store the result of the subsequence. Table 6-3 lists the name of the subproperty for each type of subsequence call.

Table 6-3. Property Names for Subsequence Results

Result Subproperty Name	Type of Subsequence Call
`TS.SequenceCall`	Sequence Call
`TS.PostAction`	Post Action Callback
`TS.SequenceFilePreStep`	SequenceFilePreStep Callback
`TS.SequenceFilePostStep`	SequenceFilePostStep Callback
`TS.ProcessModelPreStep`	ProcessModelPreStep Callback
`TS.ProcessModelPostStep`	ProcessModelPostStep Callback
`TS.StationPreStep`	StationPreStep Callback
`TS.StationPostStep`	StationPostStep Callback
`TS.SequenceFilePreInteractive`	SequenceFilePreInteractive Callback
`TS.SequenceFilePostInteractive`	SequenceFilePostInteractive Callback
`TS.ProcessModelPreInteractive`	ProcessModelPreInteractive Callback
`TS.ProcessModelPostInteractive`	ProcessModelPostInteractive Callback
`TS.StationPreInteractive`	StationPreInteractive Callback
`TS.StationPostInteractive`	StationPostInteractive Callback

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 that the step called.
- `Status`—Status of the subsequence that the step called.
- `ResultList`—Value of `Locals.ResultList` for the subsequence that the step called. This property contains the results for the steps in the subsequence.

As an example, TestStand adds the following properties to the result of any step that calls another sequence:

```
TS.SequenceCall.SequenceFile
TS.SequenceCall.Sequence
TS.SequenceCall.Status
TS.SequenceCall.ResultList
```

Loop Results

When you configure a step to loop, you can use the Record Result of Each Iteration option on the Loop tab of the Step Properties dialog box to specify that TestStand store a separate result for each loop iteration in the result list. In the result list, the results for the loop iterations come immediately after 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 that iteration. TestStand also adds the following special loop result properties to 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`.

When you run a sequence using the `TestUUTs` or `SinglePass` execution entry points, the default process model generates the test report by traversing the results for the main sequence in the client sequence file and all of the subsequences it calls. Refer to the *Process Models* section in Chapter 1, *TestStand Architecture Overview*, and to Chapter 13, *Process Models*, for more information on process models.

Engine Callbacks

TestStand specifies a set of callback sequences that it invokes at specific points during execution. These callbacks are called *engine callbacks*. TestStand defines the name of each engine callback.

Engine callbacks are a way for you to tell TestStand to 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.

The engine callbacks are in three general groups, based on the file in which the callback sequence appears. You can define engine callback sequences in normal sequence files, in process model files, and in the `StationCallbacks.seq` file.

TestStand invokes engine callbacks in a normal sequence file only when executing steps in the sequence file or loading or unloading the sequence file. TestStand invokes engine callbacks in process model files when executing steps in the model file, steps in sequences that the model calls, and steps in any nested calls to subsequences. TestStand invokes the engine callbacks in `StationCallbacks.seq` whenever executing steps on the test station.

Table 6-4 shows the different engine callbacks.

Table 6-4. Engine Callbacks

Engine Callback	Where You Define the Callback	When the Engine Calls the Callback
`SequenceFilePreStep`	Any sequence file	Before the engine executes each step in the sequence file
`SequenceFilePostStep`	Any sequence file	After the engine executes each step in the sequence file
`SequenceFilePreInteractive`	Any sequence file	Before the engine begins an interactive execution of steps in the sequence file

Table 6-4. Engine Callbacks (Continued)

Engine Callback	Where You Define the Callback	When the Engine Calls the Callback
`SequenceFilePostInteractive`	Any sequence file	After the engine completes an interactive execution of steps in the sequence file
`SequenceFileLoad`	Any sequence file	When the engine loads the sequence file into memory
`SequenceFileUnload`	Any sequence file	When the engine unloads the sequence file from memory
`ProcessModelPreStep`	Process model file	Before the engine executes each step in the process model, each step in any sequence that the process model calls, and each step in any resulting subsequence calls
`ProcessModelPostStep`	Process model file	After the engine executes each step in the process model, each step in any sequence that the process model calls, and each step in any resulting subsequence calls
`ProcessModelPreInteractive`	Process model file	Before the engine begins interactive execution of steps in a client sequence file
`ProcessModelPostInteractive`	Process model file	After the engine begins interactive execution of steps in a client sequence file
`StationPreStep`	`StationCallbacks.seq`	Before the engine executes each step in any sequence file

Table 6-4. Engine Callbacks (Continued)

Engine Callback	Where You Define the Callback	When the Engine Calls the Callback
`StationPostStep`	`StationCallbacks.seq`	After the engine executes each step in any sequence file
`StationPreInteractive`	`StationCallbacks.seq`	Before the engine begins any interactive execution
`StationPostInteractive`	`StationCallbacks.seq`	After the engine completes any interactive execution

Note ***TestStand installs predefined station engine callbacks in the*** `StationCallbacks.seq` ***file in the*** `TestStand\Components\NI\Callbacks\Station` ***directory. You can add your own station engine callbacks in the*** `StationCallbacks.seq` ***file in the*** `TestStand\Components\User\Callbacks\Station` ***directory.***

The following are examples of how you might use engine callbacks:

- You can use the `SequenceFileLoad` callback to make sure that the configuration for external devices that the subsequence file uses occurs only once during execution. Usually, you initialize the devices that a sequence requires by creating steps in the Setup group for the sequence. However, if you call the sequence repeatedly, you can move the Setup steps into a `SequenceFileLoad` callback for the subsequence file so that they run only when the sequence file loads.
- You can 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 to accumulate data on specific steps.

Note ***If you define a*** `SequenceFilePreStep`***,*** `SequenceFilePostStep`***,*** `SequenceFilePreInteractive`***, or*** `SequenceFilePostInteractive` ***callback in a model file, the callback applies only to the steps in the model file.***

Note ***You must not define a*** `SequenceFileUnload` ***callback in the*** `StationCallbacks.seq` ***sequence file. When you do this, TestStand will hang when you shut down the TestStand engine.***

Step Execution

Depending on options you set, a step performs a number of actions as it executes. Table 6-5 lists the some of the more significant actions that the step can take in the order that the step performs them.

Table 6-5. Order of Actions That a Step Performs

Action Number	Description	Remarks
1	Allocate step result	—
2	Check run mode	—
3	Evaluate precondition	—
4	Load module if not already loaded	—
5	Evaluate Loop Initialization Expression	Only if looping
6	Evaluate Loop While Expression, skip to action 18 if `False`	Only if looping
7	Allocate loop iteration result	Only if looping
8	Call prestep engine callbacks	—
9	Evaluate Pre Expression	—
10	Call Pre Step substep for step type	—
11	Call module	—
12	Call Post Step substep for step type	—
13	Evaluate Post Expression	—
14	Evaluate Status Expression	—
15	Call poststep engine callbacks	—
16	Fill out loop iteration result	Only if looping
17	Evaluate Loop Increment Expression, return to action 5	Only if looping
18	Evaluate Loop Status Expression	Only if looping
19	Unload module if required	—

Table 6-5. Order of Actions That a Step Performs (Continued)

Action Number	Description	Remarks
20	Execute post action	—
21	Fill out step result	—

Usually, a step performs only a subset of these actions, depending on the configuration of the step and the test station. When TestStand detects a run-time error, it usually proceeds to action 21. If a run-time error occurs in a loop iteration, TestStand performs action 16 before it performs action 21.

Step Status

Every step in TestStand has a `Result.Status` property. The status property is a string that indicates the result of the step execution. Although TestStand imposes no restrictions on the values to which the step or its code module can set the status property, TestStand and the built-in step types use and recognize the values that appear in Table 6-6.

Table 6-6. Standard Values for the Status Property

String Value	Meaning	Who Sets It
`Passed`	Indicates that the step performed a test that passed	Step or code module
`Failed`	Indicates that the step performed a test that failed.	Step or code module
`Error`	Indicates that a run-time error occurred.	TestStand
`Done`	Indicates that the step completed without setting its status.	TestStand
`Terminated`	Indicates that the step called a subsequence in which execution terminated. Occurs only for sequence call steps for which the Ignore Termination option is enabled.	TestStand
`Skipped`	Indicates that the step did not execute because the run mode for the step is Skip.	TestStand
`Running`	Indicates that the step is currently running.	TestStand
`Looping`	Indicates that the step is currently running in loop mode.	TestStand

Run-Time Errors

TestStand generates a run-time error if it encounters a condition that prevents a sequence from executing. If, for example, 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. A step or its code module can explicitly generate a run-time error by setting the value of the `Step.Result.Error.Occurred` property to `True`. Usually, the step or code module also sets the values of the `Step.Result.Error.Msg` and `Step.Result.Error.Code` properties to indicate the source of the error.

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

When a step causes a run-time error, the step stops executing, and TestStand sets the status of the step to `Error`. TestStand also sets the internal status of the sequence to `Error`, and execution branches to the Cleanup step group for the sequence. If the sequence is executing as a subsequence, TestStand sets the `Result.Error.Occurred` property of the calling step to `True`. TestStand also sets the `Result.Error.Code` and `Result.Error.Msg` properties of the calling step to the values of these properties in the subsequence step that generated the run-time error. In this way, the run-time error in a subsequence becomes a run-time error in the step that invokes it. The result is that TestStand executes the Cleanup steps in all active sequences and then terminates execution.

However, if the Ignore Run-Time Errors step option is enabled for a step that causes a run-time error, TestStand does not set the internal status of the sequence that contains the step to `Error`. Instead, TestStand resets the `Error.Occurred` property of the step to `False` and execution continues normally with the next step in the sequence. The `Result.Status` property in the step that caused the run-time error retains `Error` as its value.

TestStand allows you to decide interactively how to handle a run-time error. If a step causes a run-time error and the Show Dialog On Run-Time Error option is enabled in the Execution tab of the Station Options dialog box, TestStand displays the Run-Time Error dialog box, as shown in Figure 6-9.

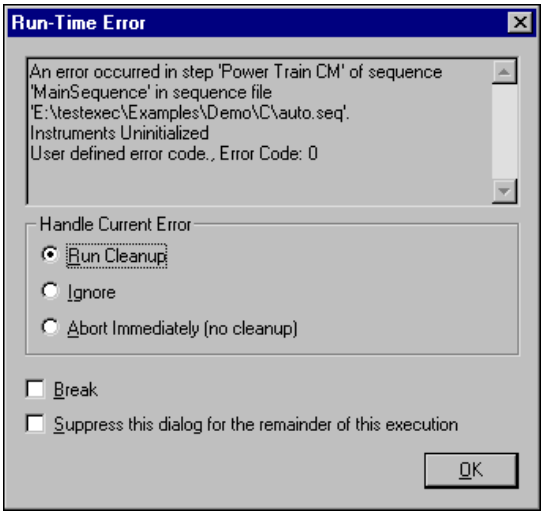

Figure 6-9. Run-Time Error Dialog Box

The Run-Time Error dialog box gives you three possible ways to handle the run-time error.

- **Run Cleanup**—The run-time error causes execution to proceed to the Cleanup step group for the sequence. This is the default action when the Show Dialog On Run-Time Error option is disabled.
- **Ignore**—TestStand does not set the internal status of the sequence to `Error`. Instead, TestStand resets the `Error.Occurred` property of the step to `False` and execution continues normally with the next step in the sequence. The `Result.Status` property of the step remains set to `Error`.
- **Abort Immediately**—TestStand stops execution immediately, without running any cleanup steps.

The dialog box also provides two further options:

- **Break**—If you choose the **Run Cleanup** or **Ignore** actions, TestStand suspends execution at the step that caused the run-time error. This option is dim if you choose **Abort Immediately**.
- **Suppress this dialog for the remainder of this execution**—This option prevents the Run-Time Error dialog box from appearing for any run-time errors that occur later in the execution. Usually, you set this option if you encounter a run-time error in a subsequence and you do not want to see the dialog box again as the error propagates to the step that called the subsequence.

7

Station Global Variables

This chapter describes station global variables and the Station Globals window.

In TestStand, you can define variables with various scopes. You can define variables that are local to a sequence, global to a sequence file, and global to the test station. You can access station global variables from any step, expression, or code module. Unlike other variables, the values of global variables are saved from one TestStand session to the next. Usually, you use station global variables to maintain statistics or to represent the configuration of your test station.

Station Globals Window

You view and edit global variables in the Station Globals window of the sequence editor. You can use the **Station Globals** menu item in the sequence editor **View** menu to display the Station Globals window. Figure 7-1 shows example variables in the Station Globals window.

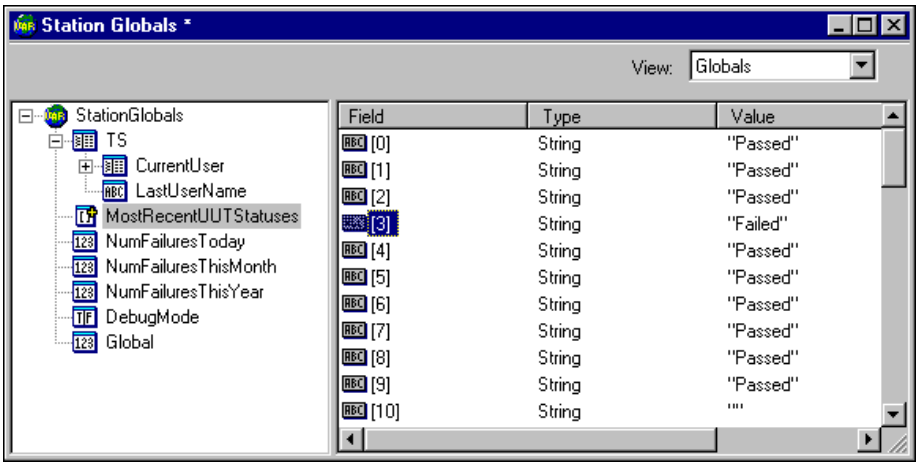

Figure 7-1. Station Globals Window

Station Globals View Ring

You can use View ring at the top right corner of the window to select which aspect of the station globals to display in the window. You have the following choices:

- **Globals**—Displays the station global variables and their values.
- **Global Types**—Displays the named data types that the station global variables use. The view is empty if no station globals use named data types. Refer to Chapter 9, *Types*, for more information on types and type editing.

Globals View Context Menu

You can display a context menu by right-clicking on a global variable, subproperty, or on the background area of the view. The context menu can contain the following items.

Insert Global

The **Insert Global** menu item has a submenu in which you select the data type for the global variable you want to insert. Figure 7-2 shows the **Insert Global** submenu.

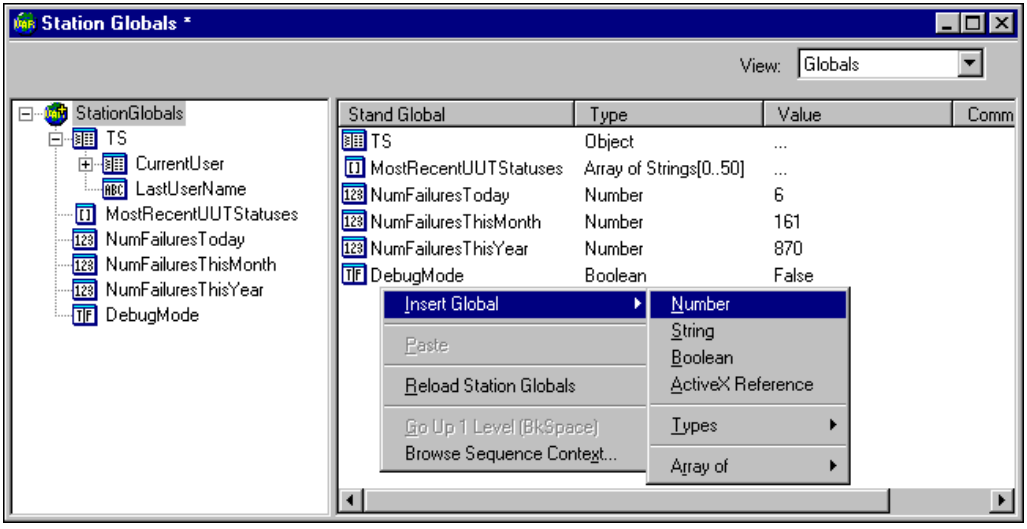

Figure 7-2. The Insert Global Submenu

If you want to insert a global variable with a custom data type, you must create a named data type first. You can create a named data type in the Global Types view of the Station Globals window, in the Sequence File Types view of a Sequence File window, or in the Type Palette window. Refer to Chapter 9, *Types*, for more information on types and type editing. After you create the named data type, it appears in the **Types** submenu of the **Insert Globals** submenu.

View Contents

The **View Contents** command selects the tree view node that corresponds to the currently selected item in the list view. The list view then displays the contents of the item. If the tree view is currently closed, it opens to show the selected node. You can use this command to view the subproperties of global variables.

Go Up One Level

The **Go Up One Level** command selects the next higher level node in the tree view. The list view displays the contents of the newly selected node.

Browse Sequence Context

The **Browse Sequence Context** command displays a tree view that contains the names of global variables and run-time state properties you can access from any expression or step module. This command also appears in the **View** menu of the sequence editor menu bar. Refer to the *View Menu* section in Chapter 4, *Sequence Editor Menu Bar*, for more information.

Rename

The **Rename** command allows you to modify the name of the selected global variable or subproperty.

Global Variable Properties

The **Properties** command displays a dialog box you can use the change the value for a global variable or one of its subproperties.

Reload Station Globals

The **Reload Station Globals** command discards the current station globals and reloads the station globals from disk. Usually, you use this command to discard edits that you make to the station globals. If you do not reload the station globals, your edits are saved when you exit the sequence editor. You can select this command only when no executions are running.

Persistence

The values of station globals persist from one TestStand session to the next. TestStand stores the station globals in the `StationGlobals.ini` file in the `Cfg` subdirectory of your TestStand engine installation. TestStand loads the station globals from `StationGlobals.ini` when the engine initializes, and it saves the station global variables to `StationGlobals.ini` when the engine shuts down. Usually, the engine initializes when you start the sequence editor or an operator interface program and the shuts down when you exit the sequence editor or operator interface. When the engine saves the station globals, not only does it save their most recent values, but it also saves any additions or deletions that you made during the session to the list of station globals.

You can save the station globals manually in the sequence editor by selecting the **Save** command in the **File** menu when the Station Globals window is the active window. You also can save the station globals to disk in a code module by using the `CommitGlobalsToDisk` method in the Engine class of the TestStand ActiveX API. When TestStand saves the station globals, TestStand changes the disk date of `StationGlobals.ini` only if the station globals in memory differ from the station globals in the file.

If you run multiple concurrent executions in the same TestStand session, all executions share the same station globals.

If you run multiple, concurrent instances of the TestStand engine, each instance maintains a separate copy of the station globals. When you start each new instance of the sequence editor or operator interface, the engine loads a copy of the station globals from `StationGlobals.ini`. If you or the sequences you run make changes to the station globals, the engine saves the current state of the station globals to `StationGlobals.ini` when the sequence editor or operator interface exits. If you make changes to the station globals in two concurrent sequence editor or operator interface instances, the instance that exits last might overwrite the changes that the other instance saved in `StationGlobals.ini`. If, when an instance exits,

the engine detects that the another instance modified the file after the current instance loaded it, the engine displays a prompt giving you the choice of overwriting the `StationGlobals.ini` or discarding your changes.

Special TestStand Station Globals

TestStand provides a special-purpose station global variable named `TS`. TestStand uses the `TS` variable to contain special values that it adds as subproperties. TestStand adds the following special variables:

- `TS.LastUserName`—A string property that holds the login name of the last user to log in.
- `TS.CurrentUser`—A property of type `User` that contains information about the user that is currently logged in. You can use the subproperties of `TS.CurrentUser` to determine if the current user has a specific privilege. Refer to the *Verifying User Privileges* section in Chapter 11, *User Management*, for more information on verifying user privileges. TestStand does not save the `TS.CurrentUser` property in `StationGlobals.ini`.

8

Sequence Context and Expressions

This chapter describes the properties in the TestStand sequence context and how to use expressions in TestStand.

Sequence Context

Before executing the steps in a sequence, TestStand creates a run-time copy of the sequence. This allows TestStand to maintain separate local variable and step property values for each sequence invocation. TestStand also maintains a *sequence context* that contains references to all global variables and to all local variables and step properties in all active sequences. The contents of the sequence context changes depending on the currently executing sequence and step.

You can use the sequence context to access variables and step properties in expressions and through calls to the TestStand ActiveX API from step modules. Refer to the *Expressions* section later in this chapter for information on expressions. For more information on the TestStand ActiveX API, refer to the *TestStand ActiveX API Reference* online help.

To refer to a subproperty, you use a period to separate the name of the property from the name of the subproperty. For example, you refer to the `CurrentUser` subproperty in the TS subproperty of the `StationGlobals` property as `StationGlobals.TS.CurrentUser`.

Tables 8-1 through 8-6 list the properties in the sequence context and describe their contents. Table 8-1 lists the first-level properties. The subsequent tables list subproperties of these properties.

Table 8-1. First-Level Properties of the Sequence Context

Sequence Context Subproperty	Description
`Step`	Contains the properties of the currently executing step in the current sequence invocation. The `Step` property exists only while a step executes. It does not exist when the execution is between steps, for example, at a breakpoint.
`Locals`	Contains the sequence local variables for the current sequence invocation.
`Parameters`	Contains the sequence parameters for the current sequence invocation.
`FileGlobals`	Contains the sequence file global variables for the current execution.
`StationGlobals`	Contains the station global variables for the engine invocation. TestStand maintains a single copy of the station globals in memory. Refer to Table 8-2 for the default contents of the `StationGlobals` property.
`ThisContext`	Holds a reference to the current sequence context. You usually use this property to pass the entire sequence context as an argument to a subsequence or a step module.
`RunState`	Contains properties that describe the state of execution in the sequence invocation. Refer to Table 8-3 for the contents of the `RunState` property.

Some of the properties in the sequence context refer to objects that exist before, and persist after, the current execution. Any modifications you make to these objects affect all executions in the current TestStand session. If you save the modifications to disk, they affect future TestStand sessions. These properties include the following:

- `Station.Globals`
- `RunState.InitialSelection`
- `RunState.SequenceFile`
- `RunState.ProcessModelClient`

Sequence Context Subproperties

This section discusses sequence context subproperties.

StationGlobals

The `StationGlobals` property object contains the station global variables for the engine invocation. Each TestStand session maintains a single copy of the station global variables in memory. Any modifications you make to a station global property affect all executions in the current TestStand session and future TestStand sessions. Refer to Chapter 7, *Station Global Variables*, for more information on station global variables.

TestStand creates a `TS` subproperty in the StationGlobals property to hold the standard station global variables that TestStand defines. Table 8-2 shows the contents of the `TS` subproperty.

Table 8-2. The StationGlobals TS Subproperty in the Sequence Context

Sequence Context Subproperty	Description
`TS`	Contains the TestStand-specific station globals.
`LastUserName`	Login name of the user that logged in most recently.
`CurrentUser`	`User` object for the user that is currently logged in. The property does not exist if no user is logged in. Refer to Chapter 11, *User Management*, for more information on the `User` standard data type.

RunState

The `RunState` property object contains all the property objects that describe the state of execution in the sequence invocation. Table 8-3 shows the subproperties of the `RunState` property object.

Table 8-3. The RunState Subproperty in the Sequence Context

Sequence Context Subproperty	Description
`Engine`	Engine object in which the sequence invocation executes. Refer to the *TestStand ActiveX API Reference* online help for more information on the methods and properties of this object.
`Root`	Sequence context for the root sequence invocation. If you initiate an execution using a process model entry point, the property is the sequence context for the process model entry point. For example, if you use an entry point from the default TestStand process model, the `Root` property is the sequence context of the `Test UUTs` or the `Single Pass` sequence. If you initiate an execution on a sequence without using a process model entry point, the `Root` property object is the sequence context for the sequence you run.
`Main`	Sequence context for the least nested sequence that is not in a process model. If you initiate an execution using the TestStand default model entry point, the `Main` property is the sequence context of `MainSequence`. If you initiate an execution on a sequence without using a process model entry point, the `Main` property object is the sequence context for whichever sequence you run.
`ThisContext`	Reference to the current sequence context. You usually use this property to pass the entire sequence context as an argument to another sequence or a step module.
`Caller`	Sequence context for the sequence that called the current sequence. This property does not exist in the root sequence context.

Table 8-3. The RunState Subproperty in the Sequence Context (Continued)

Sequence Context Subproperty	Description
`InitialSelection`	Contains references to the non-execution versions of the steps, sequences, and sequence file that are selected or active when you start the execution. You usually use this property in custom **Tools** menu commands to operate on the selected objects in a sequence file. Refer to Table 8-6 for the contents of the `InitialSelection` property. **Note:** Any changes you make to subproperty values in a Step, Sequence, or SequenceFile object that the `InitialSelection` property contains modifies the non-execution version of the object. TestStand saves the modifications when you save the selected sequence file. Whenever you modify the selected file or objects it contains from a code module, you must increment the `SelectedFile.ChangeCount` subproperty of `InitialSelection`.
`Report`	Report object for the execution. Refer to the *TestStand ActiveX API Reference* online help for more information on the methods and properties of Report objects.
`Execution`	Execution object in which the sequence invocation runs. Refer to the *TestStand ActiveX API Reference* online help for more information on the methods and properties of Execution objects.
`Thread`	Thread object in which the sequence invocation executes. Refer to the *TestStand ActiveX API Reference* online help for more information on the methods and properties of Thread objects.
`SequenceFile`	Run-time copy of the SequenceFile object for the sequence invocation. Refer to the *TestStand ActiveX API Reference* online help for more information on the methods and properties of SequenceFile objects. Refer to Table 8-4 for the contents of the `SequenceFile` property. **Note:** TestStand saves any changes you make to property values of a SequenceFile object when you save the sequence file.

Table 8-3. The RunState Subproperty in the Sequence Context (Continued)

Sequence Context Subproperty	Description
`Sequence`	Run-time copy of the Sequence object for the sequence invocation. The Sequence object contains the parameters, local variables, and steps for the sequence. Any changes you make to property values in this object modify only the execution version of the object. Refer to Table 8-5 for the contents of the `Sequence` property.
`PreviousStep`	Run-time copy of the Step object for the previously executed step in the sequence invocation. The property exists only after the first step in a step group executes. Any changes to property values in this object modify only the execution version of the object.
`Step`	Run-time copy of the Step object for the step that is currently executing. This property does not exist when the execution is between steps, for example, at a breakpoint. Any changes to property values in this object modify only the execution version of the object.
`NextStep`	Run-time copy of the Step object for the step that follows the currently executing step in the sequence. This property does not exist during and after the execution of the last step in a sequence step group. Any changes to property values in this object modify only the execution version of the object.
`IsProcessModel`	Boolean that indicates whether the sequence invocation is a sequence in the process model.
`Tracing`	Boolean that indicates whether tracing is active for the sequence invocation.
`SequenceFailed`	Boolean that indicates whether the current status of the sequence invocation is `"Failed"`.
`StepGroup`	String that contains the name of the step group that the sequence invocation is executing. Can be `"Main"`, `"Setup"`, or `"Cleanup"`.
`CallStackDepth`	Zero-based index of the currently executing sequence on the call stack. If, for example, the call stack contains three sequence invocations, `CallStackDepth` is 2. The sequence call stack includes calls to process model sequences, including calls to entry points.

Table 8-3. The RunState Subproperty in the Sequence Context (Continued)

Sequence Context Subproperty	Description
`PreviousStepIndex`	Zero-based index of the previously executed step in the step group. TestStand sets the property value to `-1` before executing the first step in a sequence step group.
`StepIndex`	Zero-based index of the currently executing step in the step group. TestStand sets the value to `-1` when the execution is between steps, such as at a breakpoint.
`NextStepIndex`	Zero-based index of the step that follows the currently executing step in the step group. TestStand sets the value to `-1` when executing the last step in a sequence step group. By modifying the value of this property, you can specify the step that TestStand executes next. **Note:** Changes that you make to this property do not affect the value of the `RunState.NextStep` property object immediately.
`LoopIndex`	The loop index for the active step in the sequence invocation. By default, steps that you configure to loop use this property to store the loop index. The value of the loop index depends on the looping construct you choose to use for the step.
`LoopNumPassed`	Number of iterations that a looping step completes with a status of `"Passed"` or `"Done"`.
`LoopNumFailed`	Number of iterations that a looping step completes with a status of `"Failed"`.
`ProcessModelClient`	The SequenceFile object for the client sequence of the process model. This property exists only for executions that you initiate through a process model entry point. Refer to the *TestStand ActiveX API Reference* online help for more information on the methods and properties of this object. **Note:** TestStand saves any changes you make to property values in the sequence file object when you save the sequence file.
`IsEditor`	Boolean that indicates whether the current GUI is a sequence editor.

RunState.SequenceFile and Other SequenceFile Objects

Several sequence context subproperties are SequenceFile objects. The subproperties are the following:

- `RunState.SequenceFile`
- `RunState.ProcessModelClient`
- `RunState.InitialSelection.SelectedFile`

Table 8-4 shows the subproperties of the SequenceFile objects. TestStand saves any changes that you make to property values in a SequenceFile object when you save the sequence file. Refer to the *TestStand ActiveX API Reference* online help for more information on the methods and properties of SequenceFile objects.

Table 8-4. The Subproperties of the SequenceFile Objects in the Sequence Context

Sequence Context Subproperty	Description
`ChangeCount`	The number of changes you have made to the sequence file object. You must increment this property whenever you modify other `SequenceFile` object properties from a code module. This indicates to the sequence editor that you have changed the sequence file.
`LastSavedChangeCount`	The value of the `ChangeCount` property when the sequence editor last saved the sequence file
`Data`	Contains the sequences and file globals in the sequence file.
`Seq`	Subproperty of `Data`. Contains an array of all Sequence objects in the sequence file.
`FileGlobalDefaults`	Subproperty of `Data`. Contains the sequence file globals variables along with their default values.
`Path`	The absolute pathname of the sequence file.

RunState.Sequence and Other Sequence Objects

Each SequenceFile object in the sequence context contains an array of Sequence objects in its `Data.Seq` subproperty. The `RunState.Sequence` subproperty of the sequence context is the Sequence object for the current sequence invocation. `RunState.Sequence` is a run-time copy of the `RunState.SequenceFile.Data.Seq` array element for the sequence that is currently executing.

Table 8-5 shows the subproperties of the Sequence objects. Refer to the *TestStand ActiveX API Reference* online help for more information on the methods and properties of Sequence objects.

Table 8-5. The Subproperties of the Sequence Objects in the Sequence Context

Sequence Context Subproperty	Description
`Locals`	Contains the local variables of the sequence. In `RunState.Sequence`, the local variables contain the current values in the sequence invocation. In nonexecution instances of Sequence objects, the local variables contain their default values.
`ResultList`	In `RunState.Sequence`, the `ResultList` local variable contains an array of step results for the sequence invocation. In nonexecution instances of Sequence objects, `ResultList` is empty.
`Main`	Contains an array of all Step objects in the Main step group.
`Setup`	Contains an array of all Step objects in the Setup step group.
`Cleanup`	Contains an array of all Step objects in the Cleanup step group.
`Parameters`	Contains the parameters of the sequence. In `RunState.Sequence`, the parameters contain the values that the calling sequence passes. In nonexecution instances of Sequence objects, the parameters contain their default values.

RunState.Step and Other Step Objects

Each Sequence object in the sequence context contains an array of Step objects for each step group. The `RunState.Step` subproperty of the sequence context is a Step object in the `RunState.Sequence` Sequence object. It represents the step that is currently executing.

All Step objects include custom properties and the standard `Result` subproperty, which contains the `Error`, `Status`, and `Common` subproperties. Refer to Chapter 10, *Built-In Step Types*, for more information on the step properties for each of the built-in step types. Refer to the *TestStand ActiveX API Reference* online help for more information on the methods and properties of Step objects.

The properties of the Step objects in `RunState.Sequence` contain the values for the current sequence invocation. The properties of the Step objects in the other Sequence objects in the sequence context contain their default values.

RunState.InitialSelection

The `RunState.InitialSelection` subproperty specifies the steps, sequences, and sequence file that are selected or active when you start an execution. You usually use this property in sequences that custom **Tools** menu commands or process model entry points call. Table 8-6 lists the subproperties of the `InitialSelection`.

Table 8-6. The InitialSelection Subproperty in the Sequence Context

Sequence Context Subproperty	Description
`SelectedSteps`	Contains an array of step objects that were selected when the execution started. The array is empty for non-root sequence contexts.
`SelectedSequences`	Contains an array of sequence objects that were selected when the execution started. The array is empty for non-root sequence contexts.
`SelectedFile`	Specifies the sequence file object for the active sequence file when the execution started. This property only exists in the root sequence context.

Any change you make to subproperty values in a Step, Sequence, or SequenceFile object that the `RunState.InitialSelection` property contains modifies the nonexecution version of the object. TestStand saves the modifications when you save the selected sequence file. Whenever you modify the selected file or objects it contains from a code module, you must increment the `ChangeCount` property of the `SelectedFile` subproperty.

Using the Sequence Context

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

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

Refer to the *Expressions* section later in this chapter for more information on using expressions.

During an execution, you can view and modify the values of the properties in the sequence context from the Context tab of the Execution window. The Context tab displays the sequence context for the sequence invocation that is currently selected in the Call Stack pane. You also can monitor individual variables or properties from the Watch Expression pane. Refer to the *Sequence Editor Execution Window* section in Chapter 6, *Sequence Execution*, for more information on using the Context tab and Watch Expression pane of the Execution window.

You can pass a reference to sequence context object to a step module. In step modules, you access the value of a variable or property by using `PropertyObject` methods in the TestStand ActiveX API on the sequence context. As with expressions, you must specify a path from the sequence context to the particular property or variable. Refer to Chapter 12, *Module Adapters*, for more information on how to pass the sequence context to a code module for each adapter. Refer to the *TestStand ActiveX API Reference* online help for more information on accessing the properties in the sequence context from code modules.

You can use the **Browse Sequence Context** command in the **View** menu of the sequence editor menu bar to display a tree view containing the names of variables, properties, and sequence parameters that you can access from expressions and step modules. Refer to the *View Menu* section in Chapter 4, *Sequence Editor Menu Bar*, for more information.

Expressions

In TestStand, you can use an expression to calculate a new value from the values of multiple variables or properties. In general, you can use an expression anywhere you can use a simple variable or property value. The Statement built-in step type evaluates an expression as a step in sequence. For most steps, you can specify a Pre Expression, a Post Expression, and a Status Expression in the Expressions tab of the Step Properties dialog box. TestStand executes the pre expression before executing the step module, and it executes that post expression and status expression after executing the step module.

In expressions, you can access all variables and properties in the sequence context that is active when TestStand evaluates the expression. The following is an example of an expression:

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

TestStand supports all applicable expression operators and syntax that you use in C, C++, Java, and Visual Basic. If you are not familiar with expressions in these standard languages, TestStand also provides an Expression Browser dialog box that you can access by clicking on the **Browse** button that appears next to controls that accept expressions.

Figure 8-1 shows the Expression Browser dialog box.

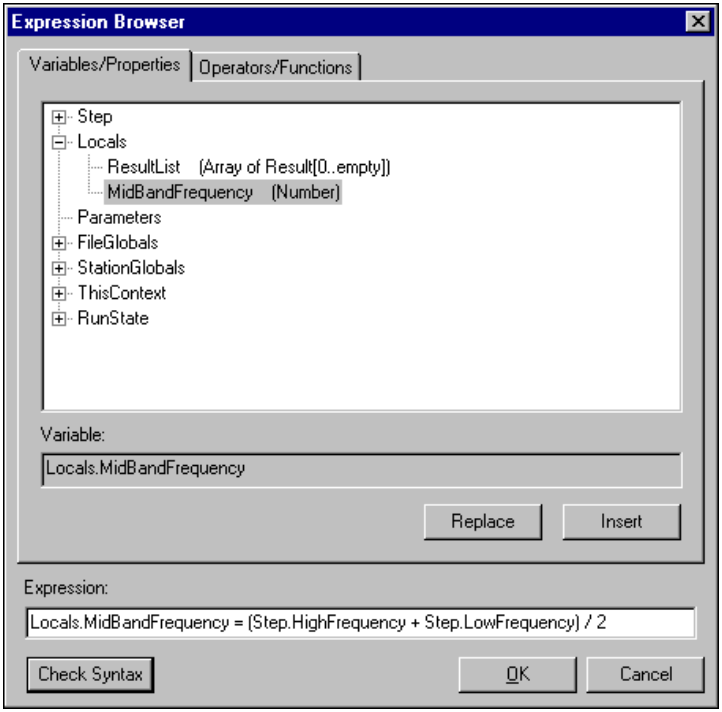

Figure 8-1. Variables/Properties Tab of the Expression Browser

The Expression Browser dialog box allows you to interactively build an expression by selecting from lists of available variables, properties, and operators. You select variables and properties from the Variables/Properties tab, and you select operators from the Operators/Functions tab. The Operators/Functions tab contains a Description text box that shows help text for the currently selected operator. Using the **Insert** and **Replace** buttons, you can copy a variable, property, or operator to the cursor location in the Expression control. Using the **Check Syntax** button, you can verify the syntax in the Expression control.

Figure 8-2 shows the Operators/Functions tab of the Expression Browser dialog box.

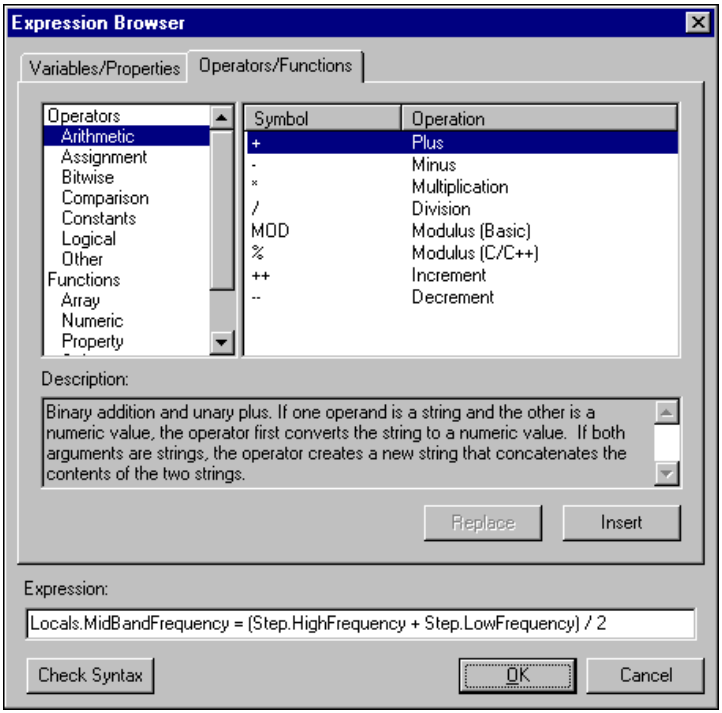

Figure 8-2. Operators/Functions Tab of the Expression Browser

Table 8-7 lists the operators and constant formats you can use in expressions.

Table 8-7. Expression Operators

Operator Class	Operators in Symbol Form
Arithmetic	The arithmetic symbols include: `+`, `-`, `*`, `/`, `MOD`, `%`, `++`, and `--`.
Assignment	The assignment symbols include: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, and `|=`.
Comparison	The comparison symbols include: `==`, `!=`, `<>`, `>`, `>=`, `<`, and `<=`.
Logical	The logical symbols include: `&&`, `||`, and `!`.
Bitwise	The bitwise symbols include: `AND`, `OR`, `NOT`, `XOR`, `&`, `|`, `~`, `^` , `>>`, and `<<`.
Constants	The formats for the different types of constants include: `1.23e-4` Floating Point `1234` Integer `0x1234efa9` Hexadecimal Integer `True` Boolean `"1234wxyz"` String Nothing Empty ActiveX Reference
Miscellaneous	Miscellaneous additional operators include: `()` Parenthesis—Alter evaluation order `.` Dot—Property field separator `[]` Brackets—Array subscript `,` Comma—Expression separator `?:` Conditional—Given a Boolean value, chooses one of two other expressions to evaluate. Usage: *`booleanValue ? expr1 : expr2`*.

The operand for an array subscript must evaluate to a numeric value, unless the array contains step or sequence elements. For arrays of step or sequence elements, the subscript can evaluate to a string value which contains the name of a step or sequence element in the array. For example, `RunState.Sequence.Main["MyGoto"]`.

Table 8-8 lists the functions you can call from an expression. Optional function parameters appear within angle brackets. For descriptions of each individual parameter, refer to the online help in the Expression Browser dialog box.

Table 8-8. Function Expression Operators

Function	Description
Array	
`GetArrayBounds(array, lower, upper)`	Retrieves the upper and lower bounds of an array.
`GetNumElements(array)`	Returns the number elements in an array.
`InsertElements(array, index, numElements)`	Inserts new elements into a one-dimensional array.
`RemoveElements(array, index, numElements)`	Removes elements from a one-dimensional array.
`SetArrayBounds(array, lower, upper)`	Changes the bounds of an array.
`SetNumElements(array, numElements)`	Sets the number of elements in a one-dimensional array.
Numeric	
`Random(low, high)`	Returns a random number between `low` and `high`.
`Round(number, <option>)`	Rounds a number to an integer.
`Val(string, <isValid>)`	Converts a string to number.
Property	
`CommentOf(object)`	Returns the comment for an object.
`NameOf(object)`	Returns the name of an object.
`PropertyExists("propertyName")`	Returns `True` if the property exists, `False` otherwise.
`TypeOf(object, <typeDisplayName>)`	Returns the type of an object.
String	
`Find(string, stringToSearchFor, <indexToSearchFrom>, <ignoreCase>, <searchInReverse>)`	Searches a string for a substring.

Table 8-8. Function Expression Operators (Continued)

Function	Description
`FindAndReplace(string, searchString, replacementString, <startIndex>, <ignoreCase>, <maxReplacements>, <searchInReverse>)`	Finds and replaces one of more substrings with a replacement string.
`Left(string, numChars)`	Retrieves a substring from the left side of a string.
`Len(string)`	Returns the number of characters in a string.
`Mid(string, startIndex,<numChars>)`	Retrieves a substring from the middle of a string.
`Replace(string, startIndex, numCharsToReplace, replacementString)`	Replaces the given number of characters at the specified index with a replacement string.
`ResStr (category, tag, <defaultString>, <found>)` or `GetResourceString(...)`	Retrieves a string from the string resource files. An alternative name for this function is `GetResourceString`.
`Right(string, numChars)`	Retrieves a substring from the right side of a string.
`Str(value)`	Converts a number or Boolean to a string.
`StrComp("StringA", "StringB", <compareOption>, <maxChars>)`	Compares two strings.
Time	
`Date(<longFormat>, <year>, <month>, <monthDay>, <weekDay>)`	Retrieves the current date.
`Time(<24Hr>, <h>, <m>, <s>, <ms>)`	Retrieves the current time.
`Seconds()`	Returns the number of seconds since the application started.
Other	
`AllOf(boolean, ...)`	Returns the logical `And` of any number of parameters.
`AnyOf(boolean, ...)`	Returns the logical `Or` of any number of parameters.

Table 8-8. Function Expression Operators (Continued)

Function	Description

`CurrentUserHasPrivilege(string)`	Returns `True` if the current user has the privilege you specify. You can specify a property path in addition to a simple property name. For example, if there is a privilege called "`Develop.SequenceFiles.Save`", then the following privileges are equivalent:
	`Develop.SaveSequenceFiles`
	`SaveSequenceFiles`

`Evaluate(string)`	Returns the value of an expression that you specify in a string.
`FindFile(file, <useCurSeqFileDir>, <PathToFile>, <promptFlag>, <searchFlag>, <canceled>)`	Attempts to locate the file you specify in the search directories.

Table 8-9 summarizes the levels of precedence in expressions.

Table 8-9. Levels of Precedence in Expressions

Expression Type	Operator	Example

primary	Literal Identifier
	(expression)
		`3.14 or "1234"`
		`Locals.String`
		`(Seconds() / 1000)`

postfix	property[index] function	`Locals.Array[25]` `Len(Locals.String)`
unary	++, --, +, -, ~, !, `NOT`	`++Locals.Number or -3.14`
multiplicative	*, /, %, `MOD`	`10 * Locals.Number`
additive	+, -	`5 - Locals.Number`
shift	<<, >>	`Locals.Number >> 2`
relational	<, >, <=, >=	`Locals.Number <= 0.1`
equality	==, <>, !=	`Locals.Number == 2.0`
bitwise AND	&, `AND`	`Locals.Number & 0xFFFF`
bitwise exclusive OR	^, `XOR`	`Locals.Number ^ 0xFFFF`
bitwise inclusive OR	|, `OR`	`Locals.Number | 0x0008`
logical AND	&&	`Locals.Bool &&` `Step.Result.PassFail`
logical OR	||	`Locals.Bool ||` `Step.Result.PassFail`
conditional	? :	`Step.Result.PassFail ? 5.0 : 6.0`
assignment	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=	`locals.number += 2.0`
comma		`Locals.Number1 = 5.0,` `Locals.Number2 = 6.0`

9

Types

This chapter discusses how you create, modify, and use step types, custom named data types, and standard named data types in TestStand. This chapter also describes the Type Palette window.

For an overview of the different categories of types, refer to the *Step Types* and *Standard and Custom Named Data Types* sections in Chapter 1, *TestStand Architecture Overview*.

Windows and Views that Display Types

The TestStand sequence editor contains four windows and views in which you can create, modify, or examine data types and step types. Each window or view displays the types that a corresponding file contains. The following list describes each window or view, its contents, and its corresponding file:

- **Sequence File Types View**—The Sequence File Types view in the Sequence File window contains tabs for the step types, custom data types, and standard data types that the variables and steps in the sequence file use. When you save the contents of the Sequence File window, TestStand writes the definitions of the types to the sequence file. Refer to Chapter 5, *Sequence Files*, for more information on the Sequence File window.
- **Station Globals Types View**—The Global Types view in the Station Globals window contains tabs for the custom data types and standard data types that the station global variables use. When you save the contents of the Station Globals window, TestStand writes the definitions of the types to the `StationGlobals.ini` file in the `TestStand\cfg` directory. Refer to Chapter 7, *Station Global Variables*, for more information on the Station Globals window.
- **User Types View**—The Types view in the User Manager window contains tabs for the custom data types and standard data types that the User objects use. All Users and User Profiles use the `User` standard data type. You can customize the User standard data type by adding subproperties to it in the Standard Data Types tab. If any of these subproperties use custom data types, the custom data types appear in the Custom Data Types tab. When you save the contents of the User

Manager window, TestStand writes the definitions to the `Users.ini` file in the `TestStand\cfg` directory. Refer to Chapter 11, *User Management*, for more information on the User Manager window.

- **Type Palette Window**—The Type Palette window contains tabs for the step types, custom data types, and standard data types that you want to have available in the sequence editor at all times. By dragging a type into the Type Palette window, you can ensure that the type is always available even when it is not in the Types views of the User Manager window, the Station Globals window, or any of the open Sequence File windows. When you save the contents of the Types Palette window, TestStand writes the definitions of the types to the `TypePalette.ini` file in the `TestStand\cfg` directory. Refer to the *Type Palette Window* section later in this chapter for more information.

Storage of Types in Files and Memory

For each type that a file uses, TestStand stores the definition of the type in the file. You also can specify that a file always saves the definition for a type, even if it does not currently use the type. Because many files can use the same type, many files can contain definitions for the same type. All your sequence files, for example, might contain the definitions for the Pass/Fail Test step type and the `CommonResults` standard data type.

In memory, TestStand allows only one definition for each type. Although the type can appear in multiple views, only one underlying definition of the type exists in memory. If you modify the type in one view, it updates in all views. The **Find Type** command in the sequence editor **View** menu displays a dialog box containing a list of all types that are currently in memory. It identifies the files and sequences that use the type. For more information, refer to the *View Menu* section in Chapter 4, *Sequence Editor Menu Bar*.

If you load a file that contains a type definition and another type definition of the same name already exists in memory, TestStand verifies that the two type definitions are identical. If they are not identical, TestStand informs you of the conflict through the Type Conflict In File dialog box.

Figure 9-1 shows the Type Conflict In File dialog box.

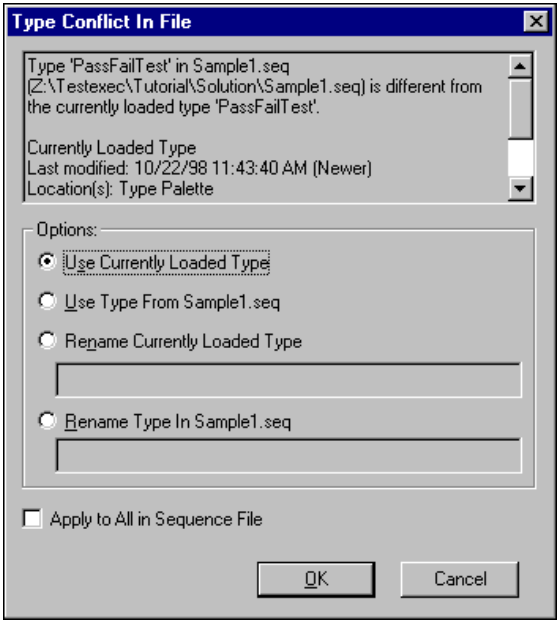

Figure 9-1. Type Conflict In File Dialog Box

You can select one of the definitions to replace the other, or you can rename one of them so that they can coexist in memory. If you enable the Apply to All in Sequence File checkbox, TestStand applies the selected option to all conflicts in the sequence file.

Using Data Types

You use data types when you insert variables, parameters, or step properties. Each view in which you can insert a variable, parameter or property has a context menu with an **Insert** item. You can use the following context menu items in the indicated views.

- **Insert Global**—Use this menu item in the Sequence File Globals view of the Sequence File window to create sequence file global variables.
- **Insert Parameter**—Use this menu item in the Parameters tab of individual sequence file views in the Sequence File window to create sequence parameters.

- **Insert Local**—Use this menu item in the Locals tab of individual sequence file views in the Sequence File window to create sequence local variables.
- **Insert Global**—Use this menu item in the Globals view of the Station Globals window to create station global variables.
- **Insert User**—Use this menu item in the Users views of the User Manager window to create new objects with the `User` data type.
- **Insert Field**—Use this menu item in the Type Palette window and the Types views in the Sequence File, Station Globals, or User Manager windows to create a new element in an existing data type.

Except for the **Insert User** item, all the preceding context menu items give you a submenu from which you can choose a data type. The submenu includes the following categories of types:

- One of the simple data types that TestStand defines. This includes the Number, Boolean, String, and ActiveX reference data types.
- A named data type. This submenu includes all the custom named data types that are currently in the Type Palette window or in the Types view of the window you are currently editing. The submenu also includes the three standard named data types that come with TestStand: `Error`, `Path`, and `CommonResults`. Refer to the *Using the Standard Named Data Types* section later in this chapter for more information.
- An array of elements that all have the same data type.

In the submenu for **Insert Parameter**, you also can select the Container type. You cannot add fields to parameters you create with the Container type. Creating a parameter with the Container type is useful only if you want to pass an object of any type to the sequence. If so, you must also turn off type checking for the parameter. If you want to create a parameter with a complex data type, you must first create the data type in the Sequence File Types view or the Type Palette window. You can then select the data type from the **Types** submenu in the **Insert Parameter** submenu.

Figure 9-2 shows the **Insert Local** submenu. The submenu includes three custom data types as examples: `Fixture`, `Subassembly` and `YieldStatistics`.

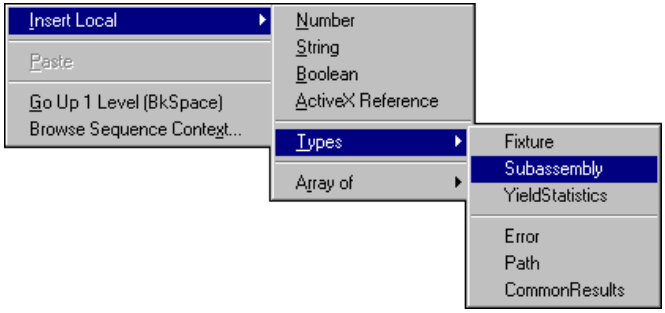

Figure 9-2. Insert Local Submenu

If the submenu does not contain the data type you require, you must create the data type in the Type Palette window or one of the type views. If the data type already exists in another window, you can drag or copy the data type from the other window to the window you are editing or to the Type Palette window.

Specifying Array Sizes

When you choose an item from the **Array of** submenu in an **Insert** submenu, the Array Bounds dialog box appears. Figure 9-3 shows the initial state of the Array Bounds dialog box.

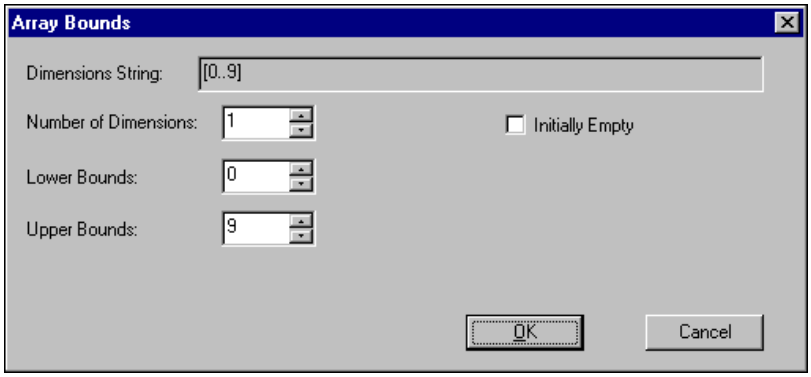

Figure 9-3. Initial State of Array Bounds Dialog Box

The Dimensions String indicator shows a string expression that describes the array dimensions. You use the Num Dimensions numeric control to set the number of dimensions in the array. The maximum number of dimensions is 16. The number of controls that appear next to the Lower Bounds and Upper Bounds labels depends on the setting of the Num Dimensions control.

You use the Lower Bounds and Upper Bounds controls to set the minimum and maximum index for each dimension. For example, you can make one dimension zero-based and another dimension one-based. The Upper Bounds setting must be greater than or equal to the Lower Bounds setting for the same dimension. You can calculate the number of elements in each dimension according to the following formula:

Upper Bounds – Lower Bounds + 1

Figure 9-4 shows the Array Bounds dialog box with settings for a three-dimensional array.

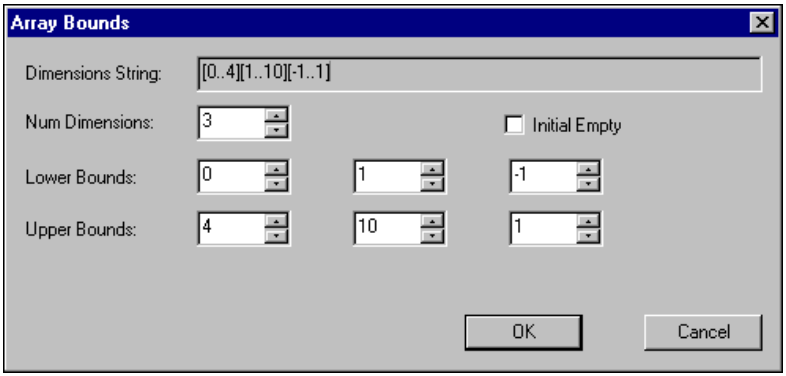

Figure 9-4. Array Bounds Dialog Box with Settings for a Three-Dimensional Array

The first and outermost dimension has 5 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 3 elements, with –1 as the minimum index and 1 as the maximum index.

After you create a variable, parameter, or property as an array, you can modify the array bounds by selecting the **Properties** item in the context menu for the variable, parameter, or property in the list view. Select the Bounds tab that now appears in the Properties dialog box to modify the array bounds.

Dynamic Array Sizing

In TestStand, you can resize an array during execution.

In an expression, you can use the `GetNumElements` and `SetNumElements` 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, you must use the `GetArrayBounds` and `SetArrayBounds` expression functions. You can find the documentation for these functions on the Operators tab of the Expression Browser dialog box. Refer to Chapter 8, *Sequence Context and Expressions*, for more information on expressions.

In a code module, you can use the `GetDimensions` and `SetDimensions` methods of the PropertyObject class to obtain or set the upper and lower bounds of an array or to change the number of dimensions. Refer to the *TestStand ActiveX API Reference* online help for more information.

Empty Arrays

If you want the array to have no elements when you start execution, enable the Initial Empty checkbox. When the Initial Empty checkbox is enabled, the Upper Bounds control for each dimension is dim. Defining an array as initially empty is useful if you do not know the maximum array size the sequence requires during execution or if you want to save memory during the periods of execution when the sequence does not use the array.

Figure 9-5 shows the Array Bounds dialog box with settings for a three-dimensional array that is initially empty.

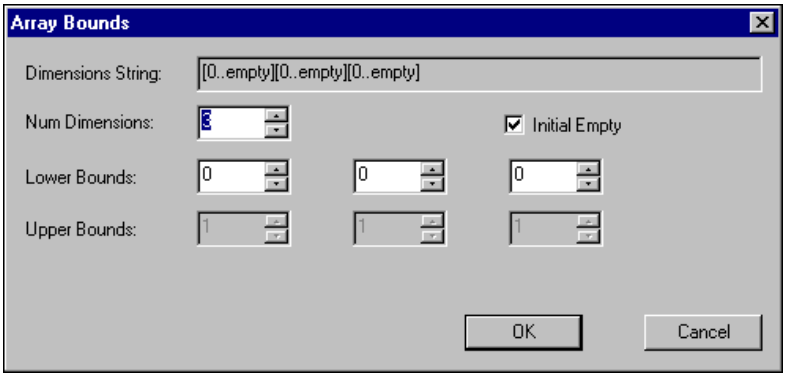

Figure 9-5. Array Bounds Dialog Box with an Initially Empty Array

Display of Data Types

The data type of each variable or property you create appears in the Type column next to the variable or property name. If the data type is an array, the words `Array of` appear in the Type column, followed by the data type of the array elements and the range of each dimension. If the data type is a named data type, the underlying type appears in the Type column, followed by the words `Instance of Type` and the data type name.

Figure 9-6 shows the variables with different data types in the Locals tab of a sequence file view.

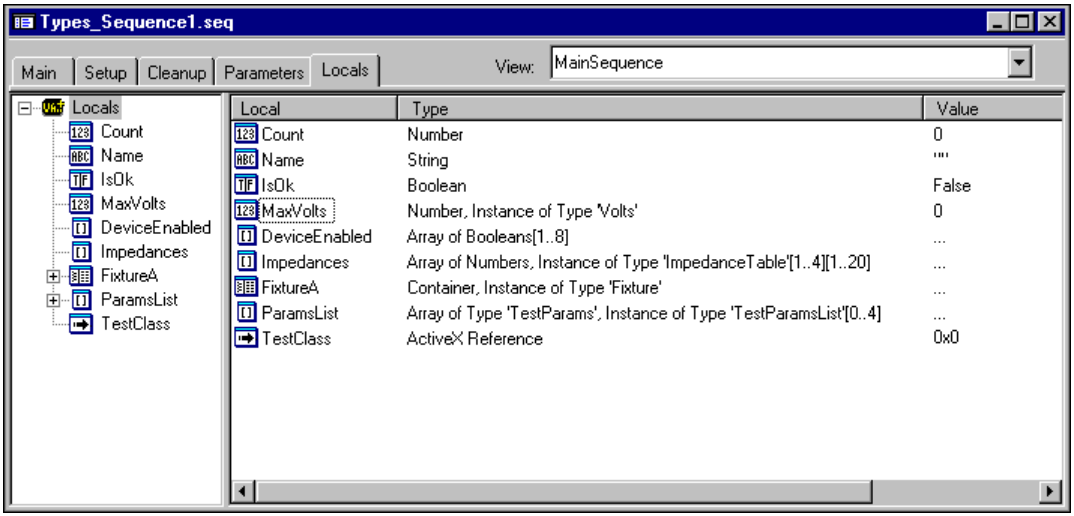

Figure 9-6. Local Variables with Various Data Types

The following describes the data type of each local variable in Figure 9-6:

- `Count` has the Number data type, which is one of the simple data types that TestStand predefines.
- `Name` has the String data type, which is one of the simple data types that TestStand predefines.
- `IsOk` has the Boolean data type, which is one of the simple data types that TestStand predefines.
- `MaxVolts` has the `Volts` data type, which is a custom data type. In this example, the `Volts` data type is simply an alias for the Number data type.
- `DeviceEnabled` is a one-dimensional array of Booleans, with indexes from 1 to 8.

- `Impedances` has the `ImpedanceTable` data type, which represents a two dimensional array of numbers.
- `FixtureA` has the `Fixture` data type, which represents a container that contains multiple fields with different data types.
- `ParamsList` has the `TestParamList` data type, which represents a one-dimensional array of elements with the `TestParams` data type. The `TestParams` data type represents a container that contains multiple fields with different data types.
- `TestClass` has the ActiveX reference data type which is one of the simple data types that TestStand predefines.

Note ***You can expand a column to the width of its largest entry by double-clicking on the vertical separator at the right edge of the column heading. This is especially useful when a variable or property has a long data type description or a long comment.***

Modifying Data Types and Values

Except for the resizing of arrays, you cannot change the internal structure of a variable, parameter, or property after you create it. You cannot change its data type setting, nor can you deviate from the data type. You can, however, change the contents of the data type itself. Changing the contents of a data type affects all variables, parameters, and properties that use the data type. Refer to the *Creating and Modifying Data Types* section later in this chapter for more information.

You can modify the value of a variable, parameter, or property in the list view in which you create it. For variables and properties, this value is the initial value when you start execution or call the sequence. For parameters, this value is the default value if you do not pass an argument value explicitly. If the data type is a single-valued data type, such as Number or Boolean, the value appears in the Value column of the list view. In Figure 9-6, the values of the first four local variables appear in the Value column.

Single Values

You modify the value of any single-valued data type, except an ActiveX reference, by selecting the **Properties** item in the context menu for the variable, parameter, or property in the list view. The Properties dialog box

appears. Figure 9-7 shows the Properties dialog box for the `Max Volts` local variable from Figure 9-6.

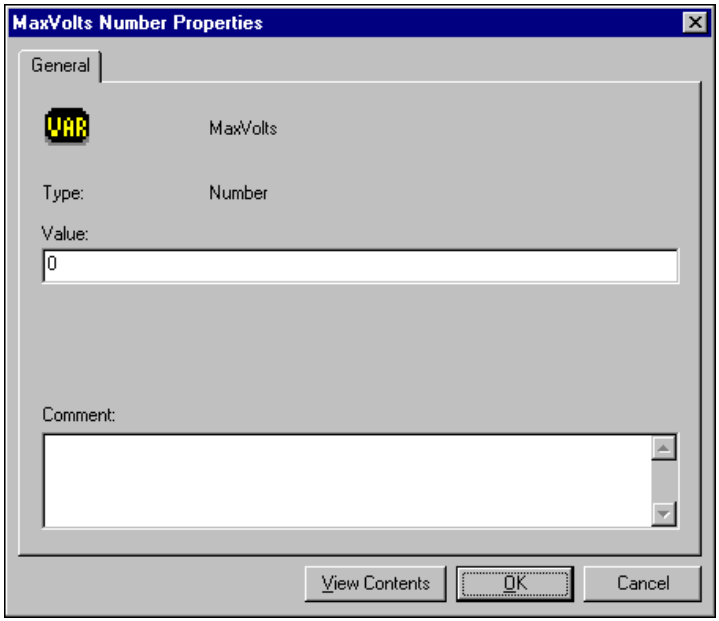

Figure 9-7. Properties Dialog Box for a Number Local Variable

If the variable, parameter, or property is an ActiveX reference, you can modify the value only from within an expression, a step code module using the TestStand ActiveX API, or by calling the TestStand ActiveX API directly using the ActiveX Automation Adapter. TestStand stores the ActiveX reference as an `IDispatch` pointer or `IUnknown` pointer. The value you assign to the ActiveX reference must be a valid ActiveX pointer. Whenever you assign an ActiveX reference a non-zero value, TestStand adds a reference to the object for as long as the variable, parameter, or property contains that value. You can release the reference to the object by assigning the variable, parameter, or property a new value or the constant `Nothing`. In addition, TestStand automatically releases the reference to the object when the variable, parameter, or property losses its scope. For example, if a sequence local variable contains a reference to an object, TestStand releases the reference when the call to the sequence completes.

Arrays

If the variable, parameter, or property is an array that contains values, you can display the elements of the array in the list view by selecting **View Contents** from the context menu. Figure 9-8 shows the contents of the `Impedances` array local variable from Figure 9-6.

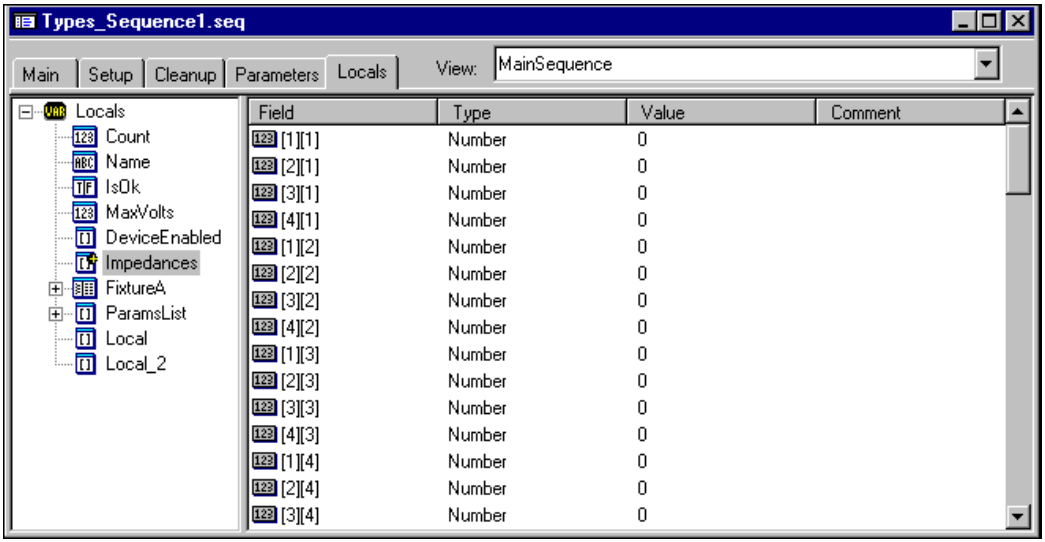

Figure 9-8. Contents of Array Local Variable in List View

The array indexes appear in the Field column of the list view. You can use the **Properties** item in the context menu for each array element to modify the initial value.

Containers

If the variable, parameter, or property is a container that contains one or more fields, you can select **View Contents** from the context menu to display the fields in the list view. For fields that have values in the Value column, you can use the **Properties** item in the context menu to examine or modify the value. For a field that is an array or container, you can select **View Contents** again to view its elements or fields.

Note ***If you want to modify an NI-installed type, you must first enable the Allow Editing NI Installed Types option in the Preferences tab of the Station Options dialog box.***

Using the Standard Named Data Types

TestStand defines three standard named data types: `Path`, `Error`, and `CommonResults`. You can add subproperties to the standard data types, but you cannot delete any of their built-in subproperties.

The Standard Data Types tab in the Type Palette window shows all three standard data types. The Standard Data Types tab in the Station Globals or Sequence File window shows only the standard data types that the variables, parameters, or properties in the window use.

Figure 9-9 shows the Standard Data Types tab of the Type Palette window.

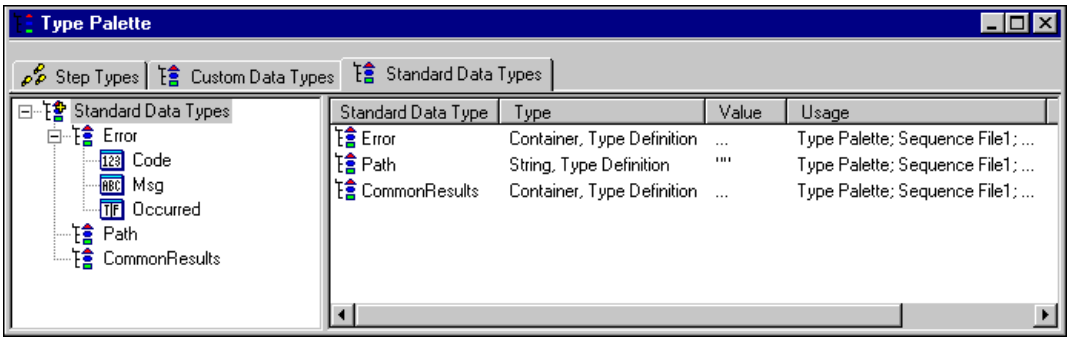

Figure 9-9. Standard Data Types Tab of the Type Palette Window

Path

You use the `Path` standard data type to store a pathname. The `Path` data type stores the pathname as a string.

The variables, parameters, and properties you define using the `Path` data type appear in the Edit Paths dialog box that the **Paths** command in the **View** menu displays. You can use the Edit Paths dialog box to view the pathnames in sequence files and station configuration files and to modify the directory portion of pathnames you select. This can be useful after you copy a sequence file or configuration file from one computer to another. The Edit Paths dialog box shows all variables, parameters, and properties that have the `Path` data type. Refer to the *View Menu* section in Chapter 4, *Sequence Editor Menu Bar*, for more information.

Error and Common Results

TestStand inserts a `Results` property in every step you create, regardless of whether you use a built-in step type or a custom step type. The `Results` property has three subproperties: `Error`, `Status`, and `CommonResults`.

The `Error` subproperty uses the `Error` standard data type. TestStand steps use the `Error` subproperty to indicate run-time errors. The `Error` standard data type is a container that contains three built-in subproperties. When a run-time error occurs in a step, the step sets the `Occurred` subproperty to `True`, the `Code` subproperty to a value that indicates that source of the error, and the `Msg` subproperty to a string that describes the error. You can add additional subproperties to the `Error` standard data type. In this way, your steps can record extra run-time error information in a standard way.

The `CommonResults` standard data type is an object that is initially empty. By adding subproperties to it, you can store extra result information for each step in a standard way.

If you choose to add additional subproperties to `Error` or `CommonResults`, newer versions of TestStand will retain them for you.

Creating and Modifying Data Types

You can create and modify data types in the Sequence File Types view of a Sequence File window, the Global Types view of the Station Globals window, and the Type Palette window. You use the Custom Data Types tab to create and modify custom data types. You use the Standard Data Types tab to add subproperties to the standard data types. The two tabs are very similar. For the sake of brevity, this section discusses creating and modifying custom data types in the Custom Data Types tab. The same information applies to the Standard Data Types tab.

Custom Data Types Tab Tree and List Views

The Custom Data Types tab contains a tree view and a list view. When you select the root node of the tree view, the custom data types appear in the list view.

Figure 9-10 shows the Custom Data Types tab for an example sequence file, with the root node selected.

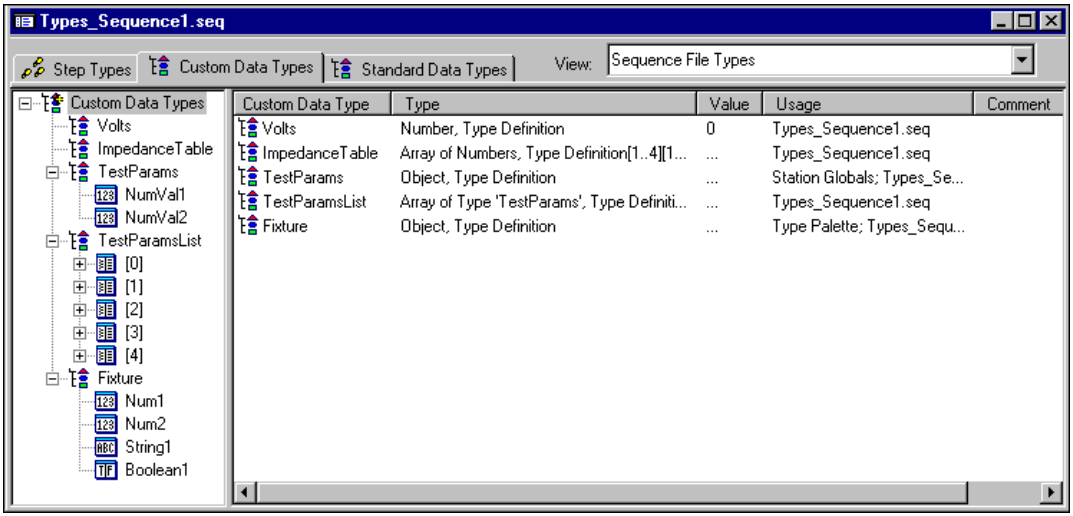

Figure 9-10. Custom Data Types Tab with Root Node Selected

The Custom Data Type column of the list view shows the name of each custom data type. The Type column shows the underlying data type. For information on the contents of the Type column, refer to the *Display of Data Types* section earlier in this chapter. If the underlying data type is a single-value type, such as Number or Boolean, the Value column shows the initial or default value that TestStand applies to all variables, parameters, and properties you create using the custom data type. The Usage column shows the files that use the data type. The Comment column shows a descriptive comment that you can create for the data type.

You can open the nodes in the tree view to show all the subproperties of each custom data type that is a container or an array of containers. You can expand a node by clicking on the plus (+) sign that appears to the left of it. When the node is open, a minus (–) sign appears to the left of the node. You can collapse the node by clicking on the minus sign.

In Figure 9-10, the tree view is partially open and shows the fields of the `TestParams` and `Fixture` containers and the elements of the `TestPararmsList` array. Notice that a plus sign appears to the left of each element of the `TestParamsList` array because each element is a `TestParams` container. You can expand an array element node to show its fields by clicking on the plus sign.

You can update the list view to display the contents of the node by selecting the node in the tree view. From the list view, you can display the contents of an item by selecting **View Contents** item from the context menu for the item. To display the contents of the next highest level, press <Backspace> in either the tree view or the list view, or select the **Go Up 1 Level** item from the context menu in the list view background.

Figure 9-11 shows the Custom Data Types tab with the list view showing the contents of the `Fixture` container data type.

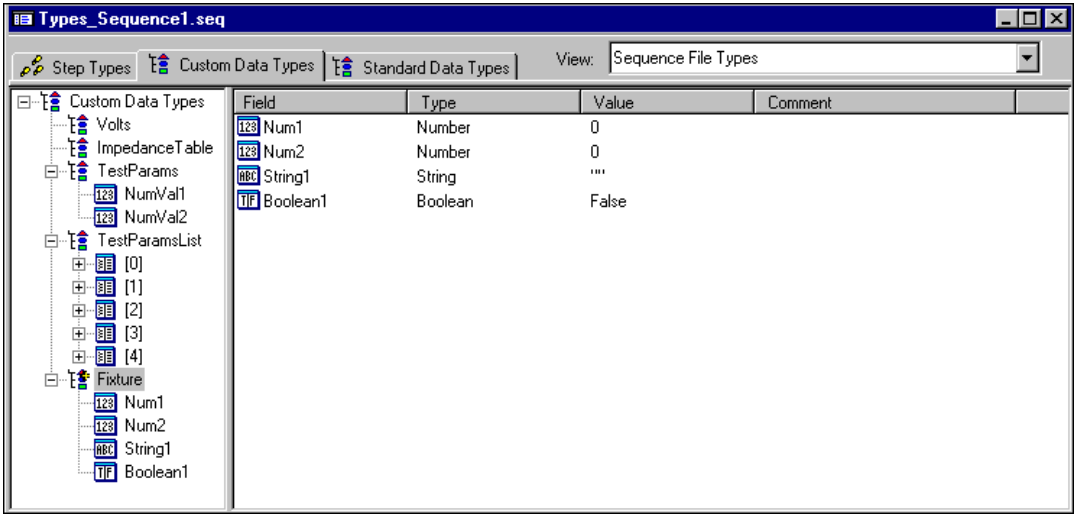

Figure 9-11. Custom Data Types Tab Showing the Contents of a Container

When you view the contents of an array, the list view displays all the array elements.

Value Field

The list view displays a value in the Value column for any item that has a single-valued type or single-valued underlying type. When you select the **View Contents** command on such an item, its Value field appears in the list view. Although you can add other fields to a single-valued data type, you usually do not. Figure 9-12 shows the Custom Data Types tab with the list view showing the Value field for the `Volts` custom data type, which uses Number as its underlying data type.

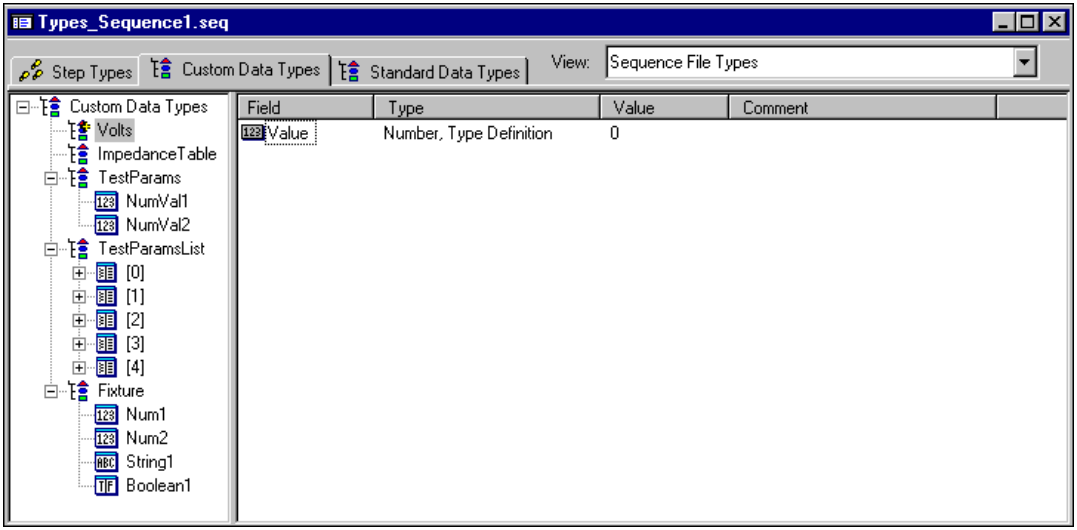

Figure 9-12. Custom Data Types Tab Showing the Value Field for a Number

If you double-click or press <Enter> on the Value field in the list view or select the **Modify Value** item from the context menu for the Value field, the Modify Value dialog box appears.

Figure 9-13 shows the Modify Numeric Value dialog box for the `Volts` data type.

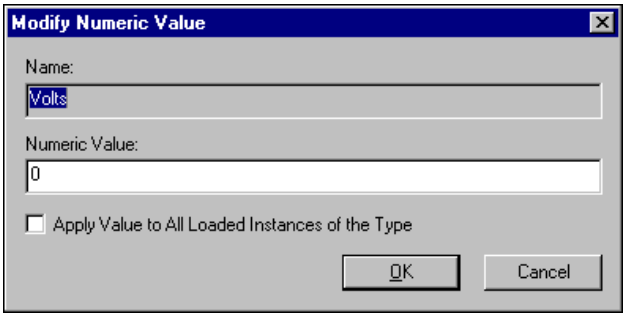

Figure 9-13. Modify Numeric Value Dialog Box

The value you specify in the dialog box is the initial or default value that all variables, parameters, or properties you create with the data type use. If any other variables, parameters, or properties already have the data type, you can change their initial or default values by enabling the Apply Value to All Loaded Instances of the Type checkbox.

Creating a New Custom Data Type

To create a new custom data type, select the root node in the tree view so that the existing custom data types appear in the list view. Right-click on the background of the list view, and select the **Insert Custom Data Type** item from the context menu. A submenu appears.

Figure 9-14 shows the **Insert Custom Data Type** submenu.

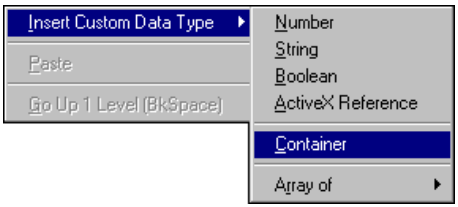

Figure 9-14. Insert Custom Data Type Submenu

The submenu gives you a set of data types from which to choose an underlying type. You can select an array of any type, a container, or any of the simple data types that TestStand defines.

If you select an array type from the submenu, the Array Bounds dialog box appears. You use the dialog box to specify the array bounds that TestStand applies initially to each variable, parameter, or property that you create with the data type. After you create the variable, parameter, or property, you can change its array bounds in the Bounds tab of the Properties dialog box. Select the **Properties** item in the context menu for the variable, parameter, or property. Refer to the *Specifying Array Sizes* section earlier in this chapter for more information on setting the size of an array.

If you select the Container type from the submenu, TestStand creates the data type without any fields.

Adding Fields to Data Types

To add fields to a new or existing data type, right-click on the icon for the data type in the list view, and select the **View Contents** item from the context menu. For a new data type, the list view becomes empty. For an existing data type, the list view displays the fields currently in the data type. Right-click on the background of the list view, and select **Insert Fields** item from the context menu. A submenu appears.

Figure 9-15 shows the **Insert Fields** submenu.

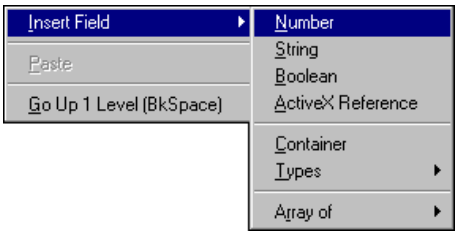

Figure 9-15. Insert Fields Submenu

The submenu gives you a set of data types to choose from. You can select any of the simple data types that TestStand defines, an array of any type, a container, or a custom or standard named data type.

You can cut, copy, paste, or rename fields using the context menu that appears when you right-click on the icon for the field in the list view.

Note ***If you want to modify a NI installed data type, you must first enable the Allow Editing NI Installed Types option in the Preferences tab of the Station Options.***

Properties Dialog Box for Custom Data Types

You can examine and modify the properties of an existing custom data type by displaying the Properties dialog box for the type. Right-click on the icon for the data type in the list view, and select the **Properties** item from the context menu. The contents of the Properties dialog box vary depending on the underlying data type.

Figure 9-16 shows the Properties dialog box for the `Volts` data type.

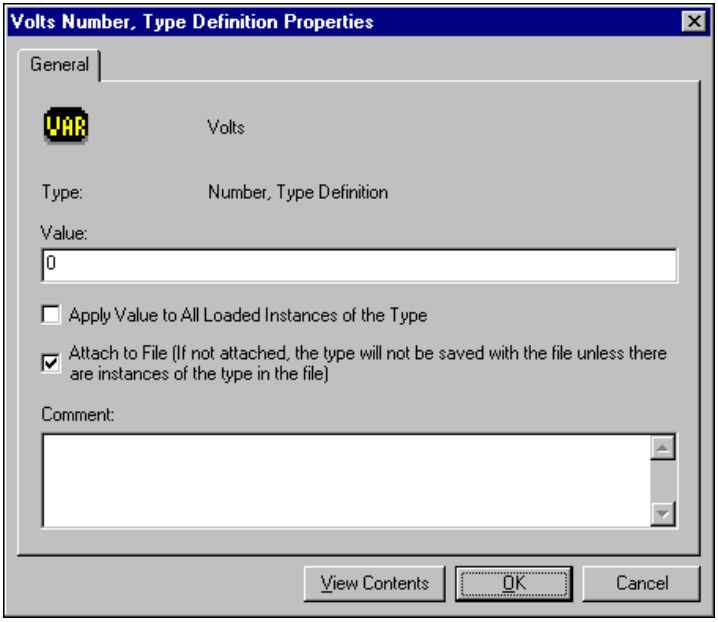

Figure 9-16. Properties Dialog Box for a Numeric Data Type

The Properties dialog box for a single-valued data type has a Value control. In the Value control, you specify the value that TestStand assigns to all variables, parameters, and properties when you create them with the data type. For variables and properties, this value is the initial value when you start execution or call the sequence. For parameters, this value is the default value if you do not pass an argument value explicitly. You can change the value in each individual variable, parameter, or property after you create it. You can assign the value you specify in the Properties dialog box for the data type to all existing instances of the data type by enabling the Apply Value to All Loaded Instances of the Type option.

The Properties dialog box for all data types have an Attach to File option. If you enable this checkbox, TestStand saves the data type in the file regardless of whether any variables, parameters, or properties in the file currently refer to it. This is useful when you design a data type and save the file before you create the variable, parameter, or property that uses the data type. When you create a new data type in the Custom Data Types tab or you copy an existing data type from another window into the tab, TestStand automatically enables the Attach to File option for you.

If you the disable the Attach to File checkbox, TestStand does not save the data type unless a variable, parameter, or property in the file refers to it. This is useful when, instead of creating or copying the data type explicitly, you copy a variable, parameter, or property that refers to the type from another window and the current file does not already contain the type. In this case, TestStand automatically disables the Attach to File option for you. If you later delete the variable, parameter, or property, TestStand also deletes the data type for you.

The Properties dialog box for an array data type contains a Bounds tab. You can change the array bounds for the data type by using the Bounds tab. Refer to the *Specifying Array Sizes* section earlier in this chapter for more information on setting the size of an array. If you have already created variables, parameters, or properties with the data type, you can change their array bounds by enabling the Apply Bounds to All Loaded Instances of the Type checkbox.

Property Dialog Box for Data Type Fields

You can examine and modify the properties for a field of a custom data type by displaying the Properties dialog box for the field. To display the fields, select the **View Contents** item from the context menu. Select the **Properties** item from the context menu for one of the fields. The Properties dialog box for a data type field is the same as the Properties dialog box for a custom data type, except that it does not contain the Attach to File checkbox.

A Properties dialog box does not exist for the Value field of a data type. Instead, you can display the Modify Value dialog box. Refer to the *Value Field* section earlier in this chapter for more information on the Modify Value dialog box.

Using Step Types

You use step types when you insert steps in the Main, Setup, and Cleanup tabs of an individual sequence view in the Sequence File window. The **Insert Step** item in the context menu displays a submenu that shows all the step types that are in the Type Palette window or the current sequence file. This includes step types that come with TestStand and custom step types you create.

Figure 9-17 shows the submenu for the **Insert Step** item. The submenu includes one custom step type, `Custom Transmitter Test`.

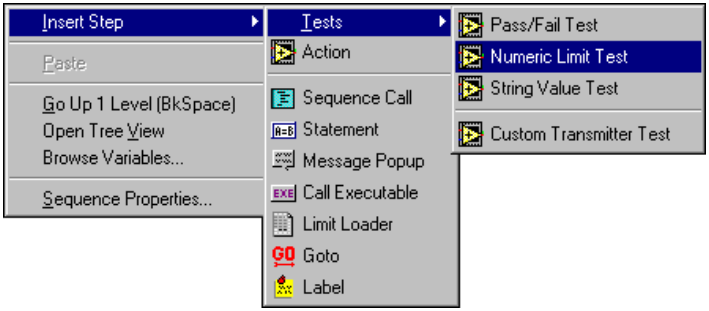

Figure 9-17. Insert Step Submenu

An icon appears to the left of each step type in the submenu. When you select a step type, TestStand displays the same icon next to the name of the new step in the list view. Many step types, such as the Pass/Fail Test and Action step types, can work with any module adapter. For these step types, the icon that appears in the submenu is the same as the icon for the module adapter that you select in the ring control on the tool bar. In Figure 9-17, the LabVIEW Standard Prototype Adapter is the current adapter, and its icon appears next to several step types, including Pass/Fail Test and Action. If you select one of these step types, TestStand uses the LabVIEW Standard Prototype Adapter for the new step.

Some step types require a particular module adapter and always use the icon for that adapter. For example, the Sequence Call step type always uses the Sequence Adapter icon. Other step types, such as Statement and Goto, do not use module adapters and have their own icons.

When you select an entry in the submenu, TestStand creates a step using the step type and module adapter that the submenu entry indicates. After you insert the step, use the **Specify Module** item in the context menu for the step to specify the code module or sequence, if any, that the step calls. The

Specify Module command displays a dialog box that is different for each adapter. Generically, the dialog box is called the Specify Module dialog box. Refer to Chapter 12, *Module Adapters*, for information on the Specify Module dialog box for each adapter. Table 9-1 shows the dialog boxes for each adapter.

Table 9-1. Adapter Dialog Box Names

Adapter	Dialog Box Title
DLL Flexible Prototype Adapter	Edit DLL Call
LabVIEW Standard Prototype Adapter	Edit LabVIEW VI Call
C/CVI Standard Prototype Adapter	Edit C/CVI Module Call
Sequence Adapter	Edit Sequence Call
ActiveX Automation Adapter	Edit Automation Call

For each step type, another item can appear in the context menu above **Specify Module**. For example, the **Edit Limits** item appears in the context menu for Numeric Limit Test steps, and the **Edit Pass/Fail Source** item appears in the context menu for Pass/Fail Test steps. The menu item displays a dialog box in which you modify step properties that are specific to the step type. This dialog box is called the *step-type-specific dialog box*. Refer to Chapter 10, *Built-In Step Types*, for information on the menu item for each of the built-in step types.

To modify step properties that are common to all step types, use the **Properties** command in the context menu, double-click on the step, or press <Enter> with the step selected. The Step Properties dialog box contains command buttons to open the Specify Module dialog box and the step-type-specific dialog box. Refer to Chapter 5, *Sequence Files*, for more information on the Step Properties dialog box.

Creating and Modifying Custom Step Types

If you want to change or enhance a TestStand built-in step type, do not edit the built-in step type or any of its supporting source code modules. Instead, copy and rename a built-in step type and its supporting modules, and make the changes to the new files. This ensures that your customizations are not lost when you install future versions of TestStand. It also makes it easier for you to distribute your customizations to other users.

The Step Types tab in the Type Palette window shows all the built-in step types. The Step Types tab in the Sequence File Types view of the Sequence File window shows only the step types that the steps in the sequence file use.

Figure 9-18 shows the Step Types tab of the Type Palette window.

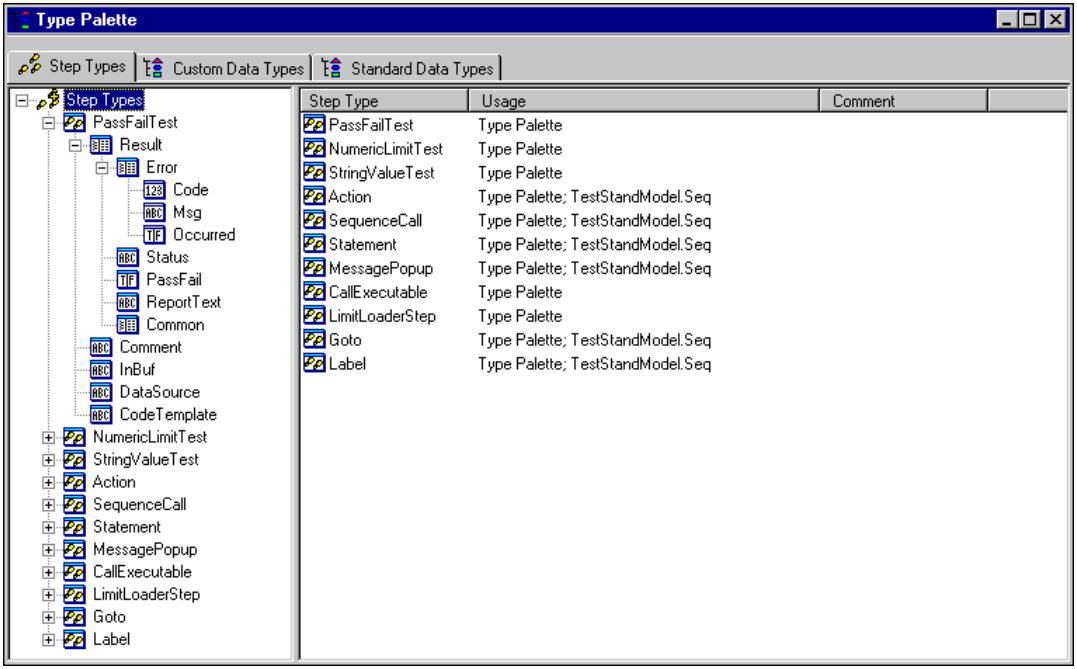

Figure 9-18. Step Types Tab of the Type Palette Window

You can insert a new step type by right-clicking on the background of the list view and selecting **Insert Step Type** item from the context menu. You can copy an existing step type by selecting the **Copy** and **Paste** items from the context menu of the step type.

Custom Step Type Properties

You can define any number of custom properties in a step type. Each step you create with the step type has the custom properties you define.

You can open the nodes in the tree view of the Step Types tab to show all step types and their custom properties. You can display the custom properties of a step type in the list view by selecting the node for the step type in the tree view. You can display the subproperties of a custom

property in the list view by selecting the node for the custom property in the tree view. From the list view, you can display the contents of a step type or property by selecting the **View Contents** item from the context menu for the step type or property. To display the contents of the next highest level, press <Backspace> in either the tree view or the list view, or select the **Go Up 1 Level** item from the context menu in the list view background.

Figure 9-19 shows the custom properties for the Numeric Limits step.

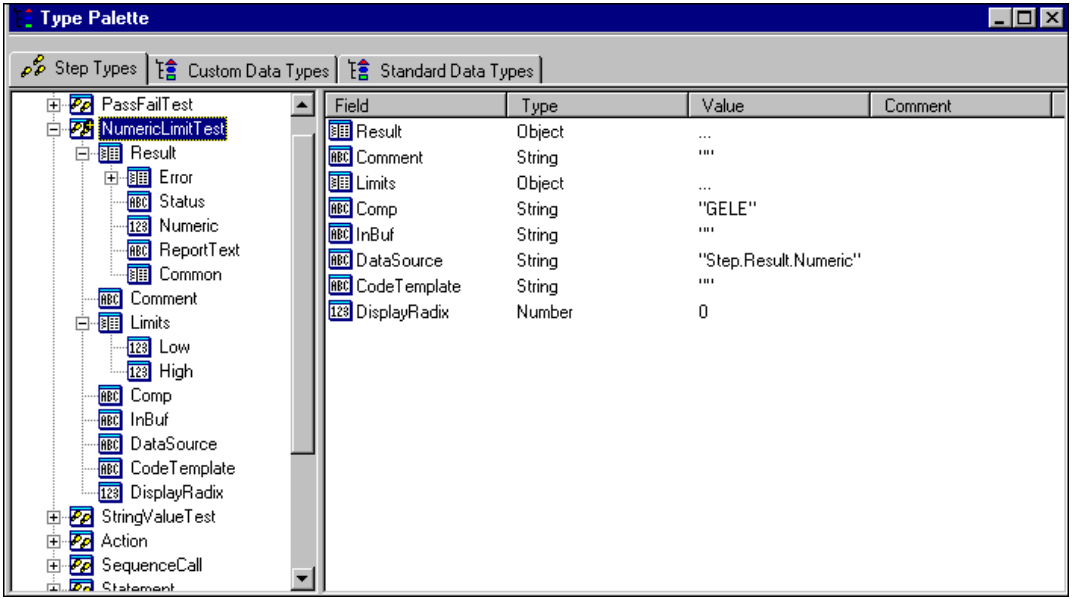

Figure 9-19. Custom Properties of a Step Type

You can add custom properties to a step type in the same way you add fields to a data type. Refer to the *Adding Fields to Data Types* section earlier in this chapter for more information.

Built-In Step Type Properties

TestStand defines many properties that are common to all step types. These are called the built-in step type properties. Some *built-in step type properties* exist only in the step type itself. These are called *class step type properties*. TestStand uses the class properties to define how the step type works for all step instances. Step instances do not contain their own copies of the class properties.

Other built-in step type properties exist in each step instance. These are called *instance step type properties*. Each step you create with the step type has its own copy of the instance properties. TestStand uses the value you specify for an instance property in the step type as the initial value of the property in each new step you create.

Normally, after you create a step, you can change the values of its instance properties. Nevertheless, when you create a custom step type, you can prevent users from changing the values of specific instance properties in the steps they create. For example, you might use the Edit substep of a step type to set the Status Expression for the step. In that case, you do not want the user to change the Status Expression value. TestStand uses this capability in some of the built-in step types, such as Numeric Limit Test and String Limit Test.

You can examine and modify the values of the built-in properties by selecting the **Properties** item from the context menu for a step type in the list view. The Step Type Properties dialog box contains the following tabs:

- General
- Menu
- Substeps
- Default Run Options
- Default Post Actions
- Default Loop Options
- Default Expressions
- Disable Properties
- Code Templates

The Default Run Options, Default Post Actions, Default Loop Options, and Default Expressions tabs display instance properties. These four tabs have the same appearance as the Run Options, Post Actions, Loop Options, and Expressions tabs of the Step Properties dialog box for a step instance. Refer to the *Step Group Context Menu* section of Chapter 5, *Sequence Files*, for more information on the Step Properties dialog box.

Most of the properties in the other five tabs are class properties. This section discusses each of these five tabs in detail.

General Tab

You use the General tab to specify a name, description, and comment for the step type. You also can specify an icon and a module adapter.

Figure 9-20 shows the General tab of the Step Type Properties dialog box for the Action step type.

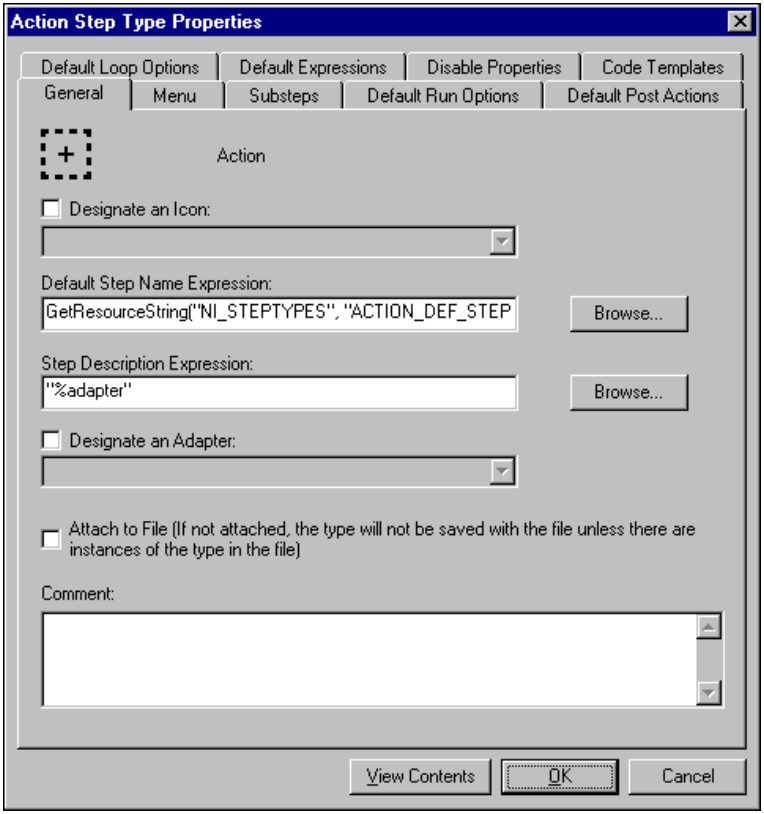

Figure 9-20. Step Type Properties Dialog Box—General Tab

The General tab of the Step Type properties dialog box contains the following controls:

- **Designate an Icon**—Use this control to specify an icon for the step type. If you enable the checkbox, you can select from a list of icons that are in the `TestStand\Components\NI\Icons` and `TestStand\Components\User\Icons` directories. TestStand displays the icon next to the step names for all steps that use the step type. If you disable the checkbox, TestStand displays the icon of the module adapter for each step. If you can use any module adapter with the step type, it is best to disable the checkbox.
- **Default Step Name Expression**—Use this control to specify a string expression that TestStand evaluates when you create a new step with the step type. TestStand uses the value of the expression as the name of the new step. If a step with the same name already exists in the sequence, TestStand appends `_Copy`*n* to the name to make it unique. If you want to store the name in a string resource file, you can use the `GetResourceString` expression function to retrieve the name from the file. Storing the name in a string resource file is useful if you want to display the name in different languages. Refer to the *Creating String Resource Files* section in Chapter 3, *Configuring and Customizing TestStand*, for more information.
- **Step Description Expression**—Use this control to specify a string expression that TestStand evaluates whenever it displays the Description field for a step. TestStand uses the value of the expression as the contents of the Description field for the step. If you include the `%adapter` macro in a string that you surround with double quotes, TestStand replaces the `%adapter` macro with text that the module adapter provides to describe the code module that the step uses.
- **Designate an Adapter**—Use this control to specify a single module adapter for the step type. If you enable the checkbox, all steps you create with the step type use the module adapter you designate, regardless of the module adapter you select in the sequence editor toolbar.

  You can choose from a list of all the TestStand module adapters. If the step type does not require a module adapter, select `<None>` from the adapter list. When you designate a module adapter, a **Specify Module** button appears. Click on the **Specify Module** button if you want to specify the module call for all steps that you create with the step type.

  If you want to prevent the sequence developer from modifying the call, enable the Specify Module checkbox in the Disable Properties tab. Refer to Chapter 12, *Module Adapters*, for information on the Specify Module dialog box for each module adapter.

- **Attach to File**—Use this control if you want TestStand to save the step type in the file regardless of whether the file contains any steps that use the step type. When you create a new step type or copy an existing step type from another window, TestStand automatically enables the Attach to File option for you.

  Disable the Attach to File checkbox if you want TestStand to save the step type only if the file contains a step that uses it. When you copy a step that uses the step type into the sequence file and the sequence file does not already contain the step type, TestStand automatically disables the Attach to File option for you. If you later delete the step, TestStand also deletes the step type for you.
- **Comment**—Use this textbox to specify text that appears in the Comment field for the step type in the list view. TestStand copies the comment into each step you create with the step type. You can change the comment for each step after you create it.

Menu Tab

You use the Menu tab to specify how the step type appears in the **Insert Step** submenu. The **Insert Step** submenu is in the context menu of individual sequence views in the Sequence File window. You can specify that the step type appears at the top-level of the **Insert Step** submenu or in a submenu you nest within the **Insert Step** submenu.

Figure 9-21 shows the Menu tab of the Step Type Properties dialog box for the Action step type.

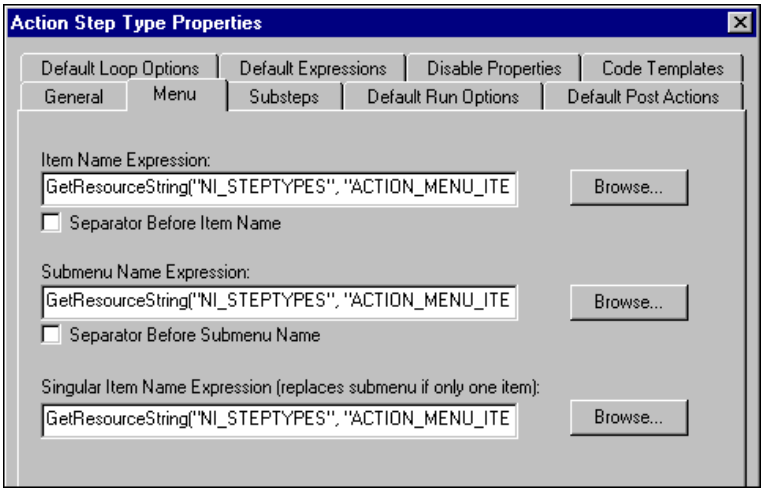

Figure 9-21. Step Type Properties Dialog Box—Menu Tab

The General tab of the Step Type properties dialog box contains the following controls:

- **Item Name Expression**—Use this control to specify an expression for the step type name that appears in the **Insert Step** submenu or in a nested submenu. If you want a separator to appear in the submenu before the name, enable the Separator Before Item Name checkbox.
- **Submenu Name Expression**—Use this control if you want to nest the step type in a submenu within the **Insert Step** submenu. If you leave this control empty or specify an empty string, the step type name appears at the top level of the **Insert Step** submenu. If you enter a nonempty expression for the submenu name, the submenu name appears at the top level of the **Insert Step** submenu, and the step type name appears in the nested submenu.

  If you want a separator to appear above the submenu name, enable the Separator Before Submenu Name checkbox. TestStand inserts a separator above the submenu name if you enable the option for at least one step type that uses the submenu name. When you change the value of the Separator Before Submenu Name checkbox, TestStand makes the same change in all step types that are currently in memory and use the same submenu name expression.
- **Singular Item Name Expression**—Use this control if you want the step type to appear at the top level of the Insert Step submenu when it is the only step type that has the submenu name you specify. Specify the step type name that you want to appear in the top level. For example, you might have a **Telecom Tests** submenu that contains **Bit Error Rate Test**, **Transmitter Test**, and **Amplifier Test** items. If the Amplifier Test is the only step type that is currently in memory, you could specify that the **Telecom Amplifier Test** item appears in the top level of the **Insert Step** submenu.

Remember that if you specify a literal string in one of the Expression controls, you must enclose it in double quotes. If want to store a name in a string resource file, you can use the `GetResourceString` expression function to retrieve the name from the file. Refer to the *Creating String Resource Files* section in Chapter 3, *Configuring and Customizing TestStand*, for more information.

Substeps Tab

You can use the Substeps tab to specify substeps for the step type. You use substeps to define standard actions, other than calling the step module, that TestStand performs for each step instance. You implement a substep through a call to a code module. The code modules you call from substeps are called *substep modules*. The *sequence developer* cannot customize the substeps for a particular step. For each step that uses the step type, TestStand calls the same substep modules with the same arguments. You can specify three substeps for a step type.

TestStand calls the *Pre Step substep* before calling the step module. You might implement a Pre Step substep to retrieve and store measurement configuration parameters into custom step properties that the step module can access.

TestStand calls the *Post Step substep* after calling the step module. You might implement a Post Step substep might to compare the values that the step module stores in custom step properties against limit values that the edit substep stores in other custom step properties.

The sequence developer can invoke the *Edit substep* by selecting a menu item that appears above the **Specify Module** item in the context menu for the step. Usually, the Edit substep displays a dialog box in which the sequence developer edits the values of custom step properties. For example, an Edit substep might display a dialog box in which the sequence developer specifies the high and low limits for a test. The Edit substep might then store the high- and low-limit values as step properties.

Figure 9-22 shows the Substeps tab of the Step Type Properties dialog box for the Numeric Limit Test step type.

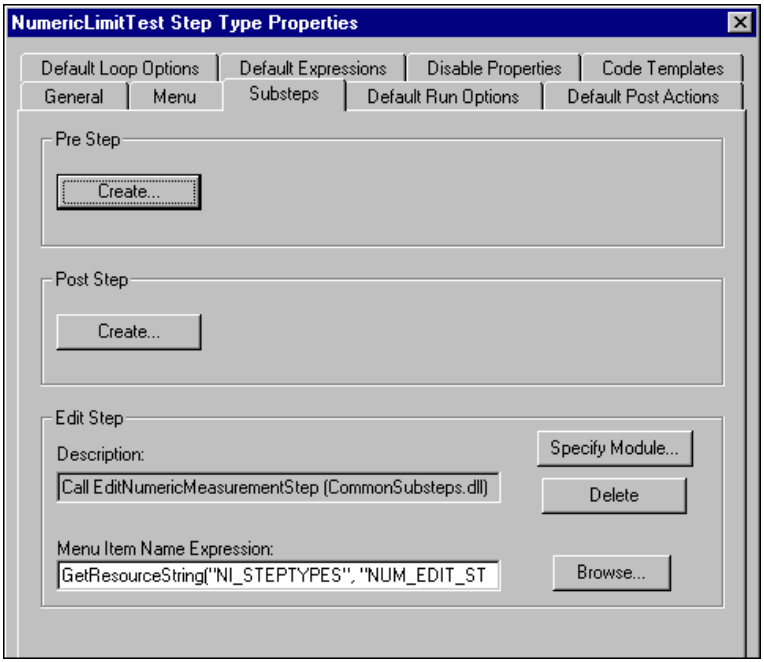

Figure 9-22. Step Type Properties Dialog Box—Substeps Tab

The Substeps tab contains a separate section for each substep. If a substep currently has no code module, its section contains only a **Create** button. If you click on the **Create** button, A dialog box appears in which you select the module adapter you want to use for the substep. You do not have to use the same module adapter that you use for the step module.

After you select the module adapter, a Description string indicator, a **Specify Module** button, and a **Delete** button appear. You use the **Specify Module** button to specify the code module to call and the parameter values to pass. Refer to Chapter 12, *Module Adapters*, for more information on the Specify Module dialog box for each module adapter.

The Description string indicator displays information about the code module for the substep. You use the **Delete** button to disassociate the code module from the substep.

The section for an Edit substep also contains a Menu Item Name Expression control. You use the control to specify an expression for the

item name that appears above the **Specify Module** item in the context menu for steps you create with the step type. The name also appears on a button on the Step Properties dialog box. If you specify a literal string for the menu item name, you must enclose it in double quotes. If you want to store the name in a string resource file, you can use the `ResStr` or `GetResourceString` expression functions to retrieve the name from the file. Refer to the *Creating String Resource Files* section in Chapter 3, *Configuring and Customizing TestStand*, for more information.

C or C++ source code is available for each of the substep modules that the built-in step types use. You can find the source code project files in the `TestStand\Components\NI\StepTypes` subdirectory. If you want to use these as starting points for your own step types, copy the files into the `TestStand\Components\User\StepTypes` subdirectory. It is best to use different filenames for the copies.

Disable Properties Tab

You can use the Disable Properties tab to prevent the sequence developer from modifying the settings of built-in instance properties in individual steps. In this way, you can make the settings you specify in the Step Type Properties dialog box permanent for all step instances.

The tab contains a list of checkboxes. Each checkbox represents one built-in instance property or a group of built-in instance properties. When you enable the checkbox, you prevent the sequence developer from modifying the value of the corresponding property or group of properties.

Figure 9-23 shows the Disable Properties tab of the Step Type Properties dialog box for the Numeric Limit Test step type.

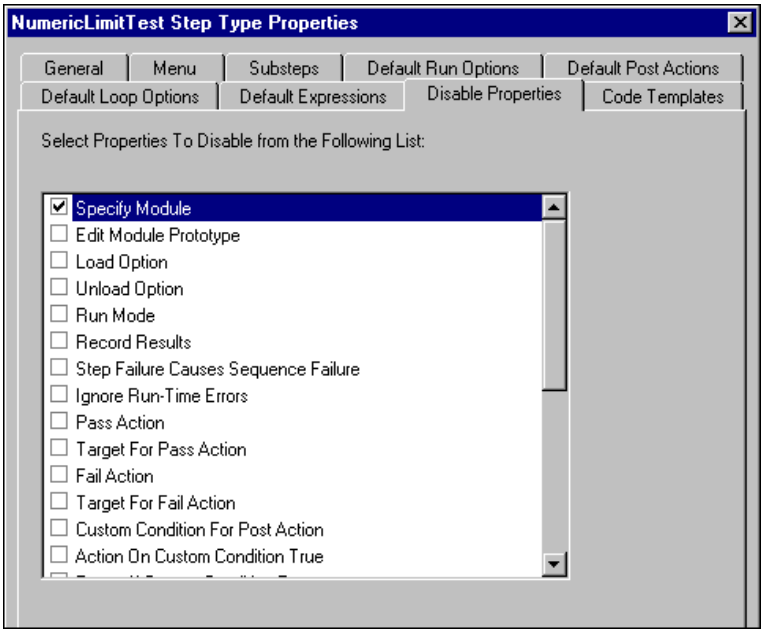

Figure 9-23. Step Type Properties Dialog Box—Disable Properties Tab

Most of the checkboxes in the Disable Properties tab apply to a specific control in the Step Properties dialog box. The two exceptions are the Specify Module checkbox and the Preconditions checkbox.

- **Specify Module**—If you check this checkbox, the sequence developer cannot display the Specify Module dialog box on any steps that use the step type. Check the Specify Module checkbox for step types that always make the same module call. Refer to the *General Tab* section earlier in this chapter for information on how to specify a module call for a step type. For example, the checkbox is unchecked for the Statement step type because Statement steps do not call code modules.

  If you uncheck the Specify Module checkbox but check the Edit Module Prototype checkbox, the sequence developer can display the Specify Module dialog box but cannot modify any of the parameter information in it.

- **Precondition**—If you check this checkbox, the sequence developer cannot create preconditions for steps that use the step type. Refer to the *Preconditions Dialog Box* section in Chapter 5, *Sequence Files*, for more information on step preconditions.

Code Templates Tab

You use the Code Template tab to associate one or more code templates with the step type. A code template is a set of source files that contain skeleton code. The skeleton code serves as a starting point for the development of code modules for steps that use the step type. TestStand uses the code template when the sequence developer clicks on the **Create Code** button on the Source Code tab in the Specify Module dialog box for a step.

TestStand comes with a default code template that you can use for any step type. You can customize code templates for individual step types. For the Numeric Limit Test step type, for instance, you might want to include example code to access the high- and low-limit properties in the step.

Templates Files for Different Adapters

Because different module adapters require different types of code modules, a code template normally consists of one or more source files for each module adapter. For the default code template, for example, TestStand comes with one `.c` file for the DLL Flexible Prototype Adapter, one `.c` file for the C/CVI Standard Prototype Adapter, and eight `.vi` files for the LabVIEW Standard Prototype Adapter. The multiple `.vi` files correspond to the different combinations of parameter options that the sequence developer can choose in the Edit LabVIEW VI Call dialog box.

TestStand uses the code template name as the name of a subdirectory in the `TestStand\CodeTemplates\NI` or `TestStand\CodeTemplates\User` directory. TestStand stores the source files for the different module adapters in the subdirectory. TestStand also stores a `.ini` file in each subdirectory. The `.ini` file contains a description string that TestStand displays for the code template. The subdirectory name for the default code template is `Default_Template`.

Code templates for the C/CVI Standard Prototype Adapter always specify two parameters: a pointer to `tTestData` structure and a pointer to a `tTestError` structure. When TestStand uses a C/CVI template module to create skeleton code, it validates the function prototype in the template module against this requirement. TestStand reports an error if the prototype is incorrect.

Code templates for the LabVIEW Standard Prototype Adapter always specify `Test Data` and `error out` clusters as parameters. The eight different `.vi` files for each LabVIEW Standard Prototype Adapter code template specify various combinations of the Input buffer, Invocation Information, and Sequence Context parameters. When TestStand uses a LabVIEW template VI to create skeleton code, it choose the correct `.vi` file to use based on the current settings in the Optional Parameters section of the Edit LabVIEW VI Call dialog box.

Code templates for the DLL Flexible Prototype Adapter can have any number of parameters that are compatible with the data types you can specify in the Module tab of the Edit DLL Call dialog box.

When TestStand uses a DLL code template source file to create skeleton code, it compares the parameter list in the source file against the parameter information in the Module tab. If they do not agree, TestStand prompts the sequence developer to select which prototype to use for the skeleton code. If the sequence developer chooses to use the prototype from the template source file, the developer also can request that TestStand update the Module tab to match the source file. The template source file does not contain sufficient information for TestStand to update the Value controls for the parameters in the Module tab.

You can specify entries for TestStand to put in the Value controls. TestStand stores this information in the `.ini` file in the template subdirectory.

Creating and Customizing Template Files

You can create a new code template in the Code Templates tab. TestStand prompts you to specify a subdirectory name and an existing code template as a starting point. TestStand copies the files for the existing template into the new subdirectory under the `TestStand\CodeTemplates\User` directory and changes the names. You must then modify the template files to customize them. If you do not intend to use a particular adapter, you can delete the template files for it.

You can customize the template files to include example code that helps the test developer learn how to access the important custom properties of the step. The method you use to customize the source files for a code template can vary based on the module adapter. For example, to show how to obtain the high- and low-limit properties in a LabVIEW or CVI template for a Numeric Limit Test step, you might include example calls to the `GetValNumber` method of the Property Class in the TestStand ActiveX API. Although you can use the `GetValNumber` method in the template for the DLL Flexible Prototype Adapter too, you might customize the prototype for the code module by specifying 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. In a LabVIEW template, you might show how to refer to the `Numeric Measurement` element of the `Test Data` cluster. In a CVI code module, you might show how to refer to the `measurement` field in the `tTestData` structure. For the DLL Flexible Prototype Adapter, you might customize the prototype in the template by specifying the measurement as a reference parameter.

Multiple 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. If a step type has multiple code templates and the sequence developer clicks on the **Create Code** button in the Specify Module dialog box, TestStand prompts the sequence developer to choose from a list of templates.

Using the Code Templates Tab

Figure 9-24 shows the Code Templates tab of the Step Type Properties dialog box for the Numeric Limit Test step type.

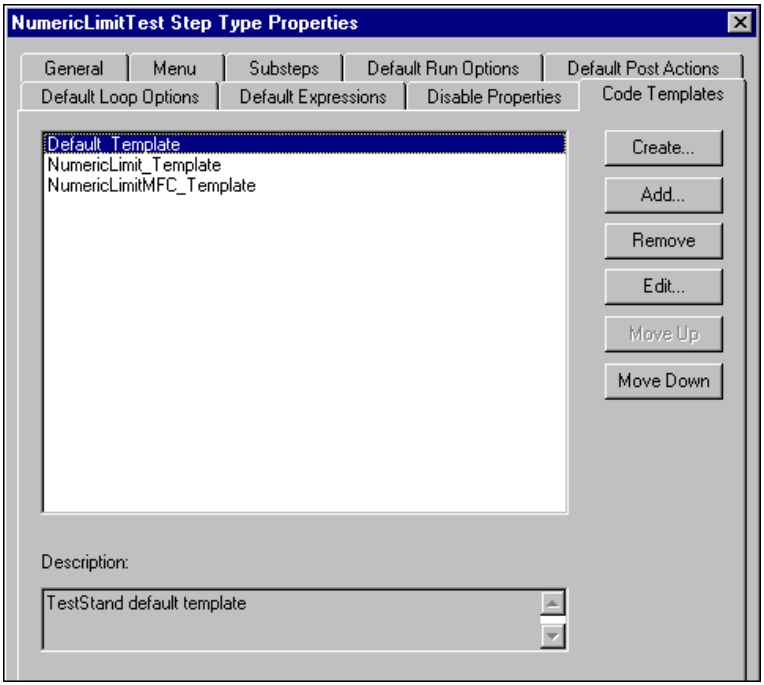

Figure 9-24. Step Type Properties Dialog Box—Code Templates Tab

The list box shows the code templates that are currently associated with the step type. The Description indicator displays the description string for the currently selected code template. The following command buttons appear to the right of the list box.

- **Create**—Use this button to create a new code template. When you click on the **Create** button, the Create Code Templates dialog box appears.

Figure 9-25 shows the Create Code Templates dialog box.

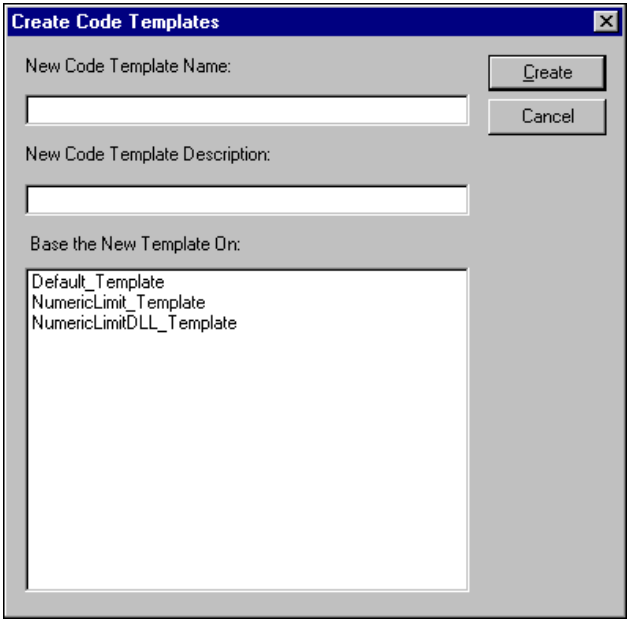

Figure 9-25. Create Code Templates Dialog Box

Specify the subdirectory name for the code template in the New Code Template Name control. Specify a brief description for the code template in the New Code Template Description control. Use the Base the New Template On list box to choose an existing code template for TestStand to copy.

- **Add**—Use this button to associate an existing code template with the step type. When you click on the **Add** button, a dialog box appears in which you can select from a list of code templates. TestStand generates the list from the set of subdirectories in the `TestStand\CodeTemplates\NI` and `TestStand\CodeTemplates\User` directories. If you specify a code template that is not in the list, the code template subdirectory must be in the TestStand search directory paths. You can customize the TestStand search directory paths with the **Search Directories** command in the **Configure** menu of the sequence editor menu bar.
- **Remove**—Use this button to disassociate the currently selected code template from the step type.
- **Edit**—Use this button to modify properties of the currently selected code template. When you click on the **Create** button, the Create Code Template dialog box appears.

- **Move Up** and **Move Down**—Use these buttons to reorder the code template list. The order of the code templates in the list box is the order that TestStand uses when displaying the code templates in the Choose Code Template dialog box.

Figure 9-26 shows the Edit Code Template dialog box.

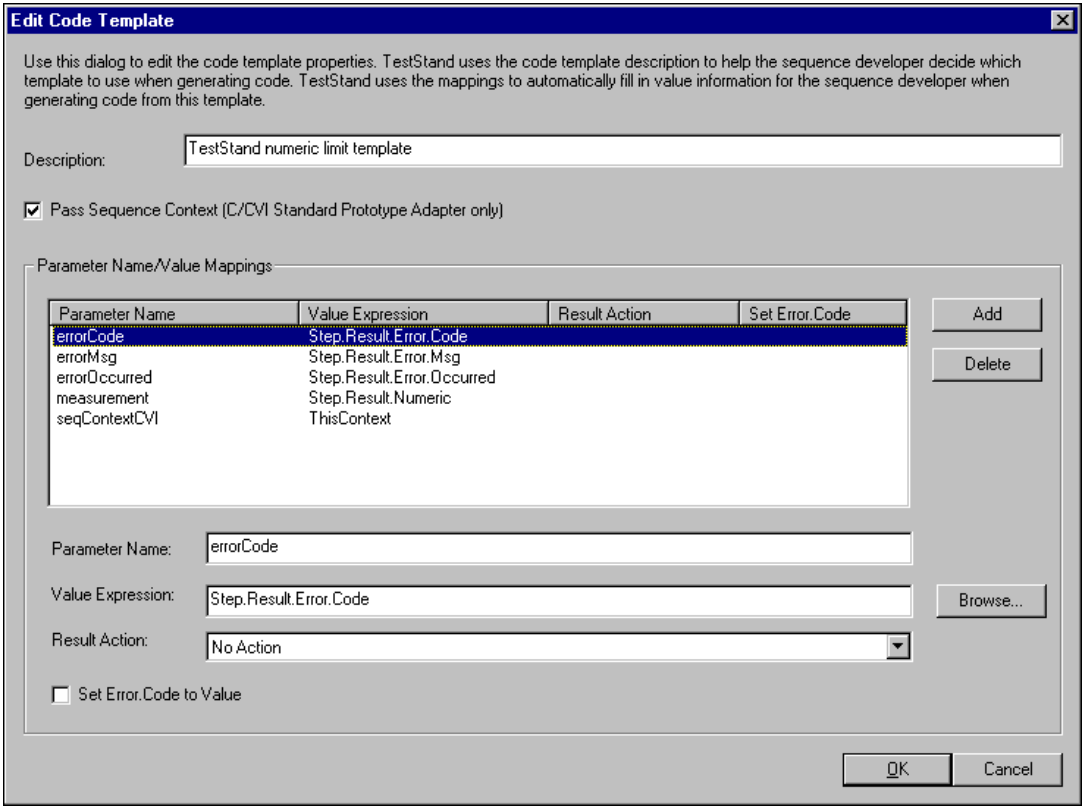

Figure 9-26. Edit Code Template Dialog Box

The Edit Code Template dialog box contains the following controls:

- **Description**—Use this control to modify the description string.
- **Pass Sequence Context**—Use this control to specify a default value for the Pass Sequence Context checkbox in the Module tab of the Specify Module box for the C/CVI Standard Prototype Adapter. TestStand applies this default value when the sequence developer clicks on the **Create Code** button. Refer to the *Specifying a C/CVI Standard Prototype Adapter Module* section in Chapter 12, *Module Adapters*, for more information.

- **Parameter Name/Value Mappings**—Use this section to specify default parameter values to use in the Module tab of the Specify Module dialog box for the DLL Flexible Prototype Adapter. TestStand applies the default parameter values when the sequence developer clicks on the **Create Code** button on the Source Code tab.

  In a template code module for the DLL Flexible Prototype Adapter, you can access step properties and sequence variables through the TestStand ActiveX API or as parameters to the code module. If you access them as parameters, the sequence developer must specify the parameter values in the Module tab. The values that the sequence developer must specify are usually the same for most step instances. You can specify default parameter values that TestStand inserts in the Module tab automatically when the sequence developer clicks on the **Create Code** button.

  The following controls are available in the Parameter Name/Value Mappings section:

  – **Add**—This button inserts an empty entry at the end of the list box.

  – **Delete**—This button deletes the currently selected entry in the list box.

  – **Parameter Name**—Enter the name of a parameter exactly as it appears in the parameter list in the template code module. To specify the return value, use `%ReturnValue`.

  – **Value Expression**—Enter the expression you want TestStand to insert into the Value control for the parameter in the Module tab of the Specify Module dialog box.

  – **Result Action**—Select the value you want to appear in the Result Action ring control of the Module tab. The sequence developer uses the ring on the Module tab to cause TestStand to set the `Error.Occurred` step property to `True` automatically when the return value or parameter value after the call is greater than zero, less than zero, equal to zero, or not equal to zero.

  – **Set Error.Code to Value**—Enable this checkbox if you want TestStand to enable the Set `Error.Code` to Value checkbox for the parameter on the Module tab of the Specify Module dialog box. This checkbox appears on the Module tab for return values and reference parameters. The sequence developer can use the checkbox on the Module tab to cause TestStand to assign the return value or output value to the `Error.Code` step property automatically.

View Contents Button

You can use the **View Contents** button on the Step Type Properties dialog box to dismiss the dialog box and show the custom properties of the step type in the list view. If you have made changes in the Step Type Properties dialog box, another dialog box appears giving you the chance to save the changes.

Type Palette Window

You use the Type Palette window to store the data types and step types that you want to be available to you in the sequence editor at all times. When you create a new type in the Sequence File Types view of a Sequence File window, the type does not appear in the **Insert Local**, **Insert Global**, **Insert Parameter**, **Insert Field**, and **Insert Step** submenus in other Sequence File windows.

To use the type in other sequence files, you can manually copy or drag the new type from one Sequence File window to another. A better approach is to copy or drag the new type to the Type Palette window or to create it there in the first place. Each type in the Type Palette window appears in the appropriate **Insert** submenus in all windows.

When you save the contents of the Types Palette window, TestStand writes the definitions of the types to the `TypePalette.ini` file in the `TestStand\cfg` directory.

The Type Palette window contains tabs for step types, custom data types, and standard data types. After you install TestStand, the Step Types tab has all the built-in step types, the Custom Data Types tab is empty, and the Standard Data Types tab has the three standard data types.

10

Built-In Step Types

This chapter describes the predefined step types that TestStand includes. The built-in step types fall into three categories: step types that can call code modules using any module adapter, step types that work with a specific module adapter, and step types that do not use module adapters at all.

Overview

Common Custom Properties

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

The built-in step types share some common custom properties. Figure 10-1 shows the common custom step properties.

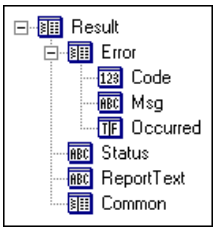

Figure 10-1. Properties That All Steps Contain

The common custom step properties are the following:

- `Step.Result.Error.Occurred` is a Boolean flag that indicates whether a run-time error occurred in the step. This document refers to this property as the *error occurred flag*.
- `Step.Result.Error.Code` is a code that describes the error that occurred.
- `Step.Result.Error.Msg` is a message string that describes the error that occurred.

- `Step.Result.Status` specifies the status of the last execution of the step, such as `Done`, `Passed`, `Failed`, `Skipped`, or `Error`. This document refers to this property as the *step status*.
- `Step.Result.Common` is a placeholder container that you can customize. You customize it by modifying the `CommonResults` standard data type. Refer to the *Using Data Types* section in Chapter 9, *Types*, for more information on standard TestStand data types.
- `Step.Result.ReportText` contains a message string that TestStand includes in the report. You can set the value of the message string directly in the code module. The C/CVI and LabVIEW module adapters allow code modules to set this property by modifying the corresponding member of the test data structure or cluster. Refer to Chapter 12, *Module Adapters*, for more information on the property assignments that the module adapters automatically perform to and from step properties.

Step Status, Error Occurred Flag, and Run-Time Errors

A code module can report a run-time error condition by setting the error occurred flag to `True`. If an exception occurs in the code module or at any other time during step execution, the TestStand engine sets the error occurred flag to `True`.

If the error occurred flag is `True` when a step finishes executing, TestStand does not evaluate the post and status expressions for the step. Instead TestStand sets the step status to `Error`. If the Ignore Run-time Errors step property is `False`, the TestStand engine reports the run-time error to the sequence. If the Ignore Run-time Errors step property is `True`, execution continues normally after the step.

Before TestStand executes a step, it sets the step status to `Running` or `Looping`. If, after the step executes, the error occurred flag is `False`, and the step status is still `Looping` or `Running`, TestStand changes the step status to `Done`. The step status is `Passed` or `Failed` only after a code module, a module adapter, or a step type explicitly sets the step status to one of these two values. Refer to Chapter 12, *Module Adapters*, for more information on the assignments that module adapters make to and from step properties.

Customizing Built-In Step Types

If you want to change or enhance a TestStand built-in step type, do not edit the built-in step type or any of its supporting source code modules. Instead, make your own copies of the built-in step type and any supporting modules,

and make the changes to these copies. This ensures that you do not lose your changes when you install future versions of TestStand.

Source code is available for the code modules that the built-in step types use as substeps. You can find the source code project files in the `TestStand\Components\NI\StepTypes` subdirectory. Make your own copies of these files in the `TestStand\Components\User\StepTypes` subdirectory and rename them.

Step Types That You Can Use with Any Module Adapter

TestStand comes with four built-in step types that you can use with any module adapter: Action, Pass/Fail Test, Numeric Limit Test, and String Value Test. When you insert a step in a sequence, TestStand binds the step to the adapter that is currently selected in the ring on the sequence editor toolbar. The icon for the adapter appears as the icon for the step. The icons for the different adapters are as follows:

C/CVI Standard Prototype Adapter

LabVIEW Standard Prototype Adapter

DLL Flexible Prototype Adapter

Sequence Adapter

ActiveX Automation Adapter

<None>

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

You specify the code module that the step calls by selecting the **Specify Module** item from the step context menu or the **Specify Module** button on the Step Properties dialog box. Each module adapter displays a different Specify Module dialog box. Refer to Chapter 12, *Module Adapters*, for more information on the Specify Module dialog box for each module adapter.

Action

You usually use Action steps to call code modules that do not perform tests but rather 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. Thus, the status for an Action step is `Done` or

`Error` unless you specifically set the status in the code module 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 that all steps contain.

Pass/Fail Test

You usually use a Pass/Fail Test step to call a code module that makes its own pass/fail determination.

After the code module executes, the Pass/Fail Test step type evaluates the `Step.Result.PassFail` property. If `Step.Result.PassFail` is `True`, the step type sets the step status to `Passed`. Otherwise, it sets the step status to `Failed`.

The following are the different ways that a code module can set the value of `Step.Result.PassFail`:

- You can set the value of `Step.Result.PassFail` directly in a code module by using the TestStand ActiveX API.
- You can pass `Step.Result.PassFail` as a reference parameter to a subsequence or code module if you use the Sequence Adapter, the DLL Flexible Prototype Adapter, or the ActiveX Automation Adapter.
- The C/CVI and LabVIEW module adapters update the value of `Step.Result.PassFail` automatically after calling the code module. The C/CVI module adapter updates the value of `Step.Result.PassFail` based on the value of the `result` field of the `tTestData` parameter that it passes to the C function. The LabVIEW module adapter updates the value of `Step.Result.PassFail` based on the value of `Pass/Fail Flag` in the `TestData` cluster that it passes to the VI. Refer to Chapter 12, *Module Adapters*, for more information on the assignments that module adapters make to and from step properties.

By default, the step type uses the value of the `Step.Result.PassFail` Boolean property to determine whether the step passes or fails. You can customize the Boolean expression the step type uses by selecting the **Edit Pass/Fail Source** item in the context menu for the step or the **Edit Pass/Fail Source** button on the Step Properties dialog box.

Figure 10-2 shows the Edit Pass/Fail Source dialog box.

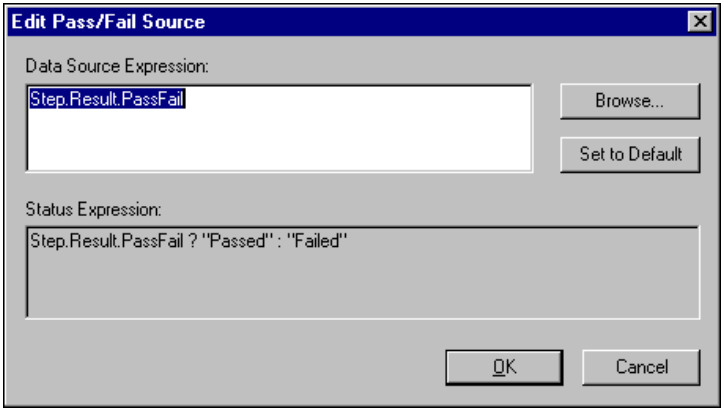

Figure 10-2. Edit Pass/Fail Source Dialog Box

Figure 10-3 shows the step properties for the Pass/Fail Test step type.

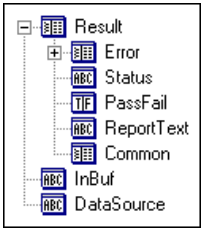

Figure 10-3. Pass/Fail Test Step Properties

The Pass/Fail Test step type defines the following step properties in addition to the common custom properties.

- `Step.Result.PassFail` specifies the Boolean pass/fail flag. Pass is `True`, Fail is `False`. Usually, you set this value in the step module.
- `Step.InBuf` specifies an arbitrary string that the C/CVI and LabVIEW module adapters pass to the test in the `tTestData` structure or `TestData` cluster automatically. 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 ActiveX API.
- `Step.DataSource` specifies the Boolean expression that 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 that the code module sets. You can customize this

expression if you do not want to set the value of `Step.Result.PassFail` in the code module. For example, you can set the data source expression to refer to multiple variables and properties, such as, `RunState.PreviousStep.Result.Numeric * Locals.Attenuation > 12`.

Numeric Limit Test

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

You can customize the type of comparison and limits that TestStand uses to set the step status. To do so, select the **Edit Limits** item from the step context menu or click on the **Edit Limits** button on the Step Properties dialog box.

Figure 10-4 shows the Limits tab on the Edit Numeric Limit Test dialog box.

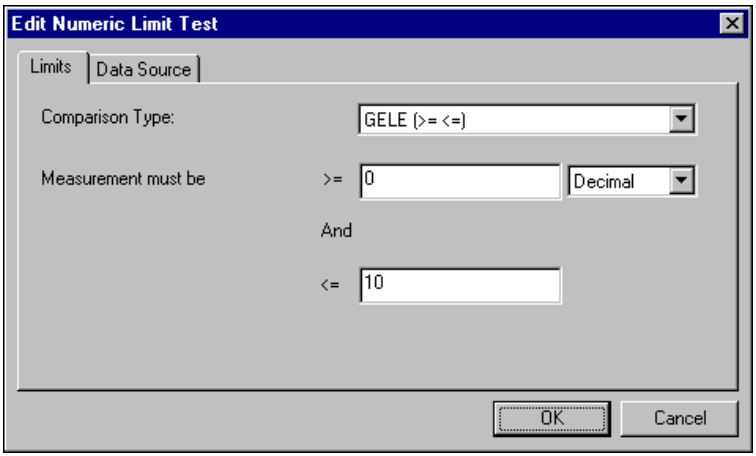

Figure 10-4. Limits Tab on Edit Numeric Limit Test Dialog Box

The Comparison Type selection ring on the Limits tab specifies the type of comparison the step type performs, if any, to determine the step status. Table 10-1 lists the available comparison types.

Table 10-1. Numeric Limit Test Comparison Types

Type	Description
`EQ`	Numeric Measurement = Low Limit
`NE`	Numeric Measurement != Low Limit
`GT`	Numeric Measurement > Low Limit
`LT`	Numeric Measurement < Low Limit
`GE`	Numeric Measurement >= Low Limit
`LE`	Numeric Measurement <= Low Limit
`GTLT`	Numeric Measurement > Low Limit and < High Limit
`GELE`	Numeric Measurement >= Low Limit and <= High Limit
`GELT`	Numeric Measurement >= Low Limit and < High Limit
`GTLE`	Numeric Measurement > Low Limit and <= High Limit
`No Comparison`	TestStand makes no Pass/Fail determination, and sets the status to Passed automatically.

Depending on the setting of the Comparison Type selection ring, the dialog box display additional controls in which you enter high and low limits. You can choose to display the limit values in decimal, hexadecimal, octal, or binary formats.

A Numeric Limit Test step always 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:

- You can set the value of `Step.Result.Numeric` directly in a code module by using the TestStand ActiveX API.
- You can pass `Step.Result.Numeric` as a reference parameter to a subsequence if you use the Sequence Adapter, the DLL Flexible Prototype Adapter, or the ActiveX Automation Adapter.
- The C/CVI and LabVIEW module adapters update the value of `Step.Result.Numeric` automatically after calling the code module. The C/CVI module adapter updates the value of `Step.Result.Numeric` based on the value of the `measurement`

field of the `tTestData` parameter that it passes to the C function. The LabVIEW module adapter updates the value of `Step.Result.Numeric` based on the value of `Numeric Measurement` in the `TestData` cluster that it passes to the VI. Refer to Chapter 12, *Module Adapters*, for more information on the assignments that the module adapters automatically makes to and from step properties.

By default, the step type uses the value of the `Step.Result.Numeric` property as the numeric measurement to compare the limits against. You can customize the numeric expression by selecting the Data Source tab of the Edit Numeric Limit Test dialog box, as shown in Figure 10-5.

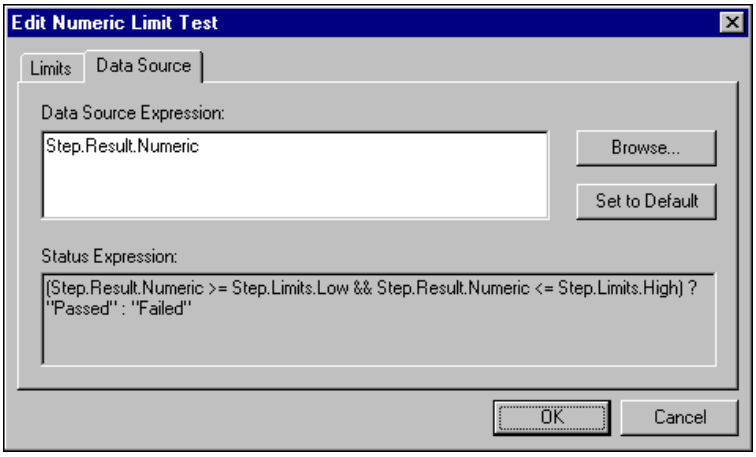

Figure 10-5. Data Source Tab on Edit Numeric Limit Test Dialog Box

Figure 10-6 shows the step properties for the Numeric Limit Test step type.

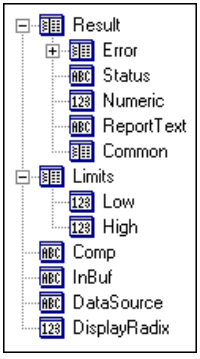

Figure 10-6. Numeric Limit Test Step Properties

The Numeric Limit Test step type defines the following step properties in addition to the common custom properties.

- `Step.Result.Numeric` specifies the numeric measurement value. Usually, you set this value in the step module.
- `Step.Limits.High` and `Step.Limits.Low` specify the limits for the comparison expression.
- `Step.Comp` specifies the type of comparison, for example, `EQ`.
- `Step.InBuf` specifies an arbitrary string that the C/CVI and LabVIEW module adapters pass to the test in the `tTestData` structure or `TestData` cluster automatically. This property exists to maintain compatibility with previous test executives. Usually, code modules that you develop for TestStand receive data as input parameters or access data as properties using the TestStand ActiveX API.
- `Step.DataSource` specifies a numeric expression that the step type uses to set value of `Step.Result.Numeric`. The default value of the expression is `"Step.Result.Numeric"`, which has the effect of using the value that the code module sets. You can customize this expression if you do not want to set the value of `Step.Result.Numeric` in the code module.
- `Step.DisplayRadix` specifies the display format for limit values.

You can use a Numeric Limit Test without a code module. This is useful if you want to limit-check a value that you already have. To do this, select <None> as the module adapter before inserting the step in the sequence, and configure `Step.DataSource` to specify the value that you already have.

String Value Test

You usually 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 that the step obtains to the string that the step expects to receive. If the string that the step obtains matches the string that it expects, the step type sets the step status to `Passed`. Otherwise, it sets the step status to `Failed`.

You can customize the type of comparison that TestStand uses to set the step status. You can also specify the string that the step expects to receive. To do so, select the **Edit Expected String** item in the context menu for the step or the **Edit Expected String** button in the Step Properties dialog box.

Figure 10-7 shows the Limits tab on the Edit String Value Test dialog box.

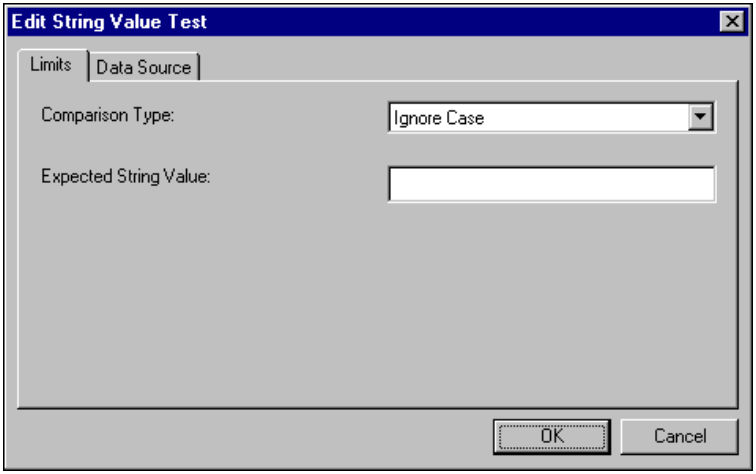

Figure 10-7. Limits Tab on the Edit String Value Test Dialog Box

On the Limits tab, you can specify the expected string and whether the string comparison is case-sensitive.

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

- You can set the value of `Step.Result.String` directly in a code module by using the TestStand ActiveX API.
- You can pass `Step.Result.String` as a reference parameter to a subsequence or code module if you use the Sequence Adapter, the DLL Flexible Prototype Adapter, or the ActiveX Automation Adapter.
- A code module can directly modify the value `Step.Result.String` using the TestStand ActiveX API. The C/CVI and LabVIEW module adapters update the value of `Step.Result.String` automatically after calling the code module. The C/CVI module adapter updates the value of `Step.Result.String` based on the value of the `stringMeasurement` field of the `tTestData` parameter that it passes to the C function. The LabVIEW module adapter updates the value of `Step.Result.String` based on the value of `String Measurement` in the `TestData` cluster that it passes to the VI. Refer to Chapter 12, *Module Adapters*, for more information on the assignments that the module adapters automatically make to and from step properties.

By default, the step type uses the value of the `Step.Result.String` property as the string value to compare the limits against. You can customize the string expression by selecting the Data Source tab of the Edit String Value Test dialog box, as shown in Figure 10-8.

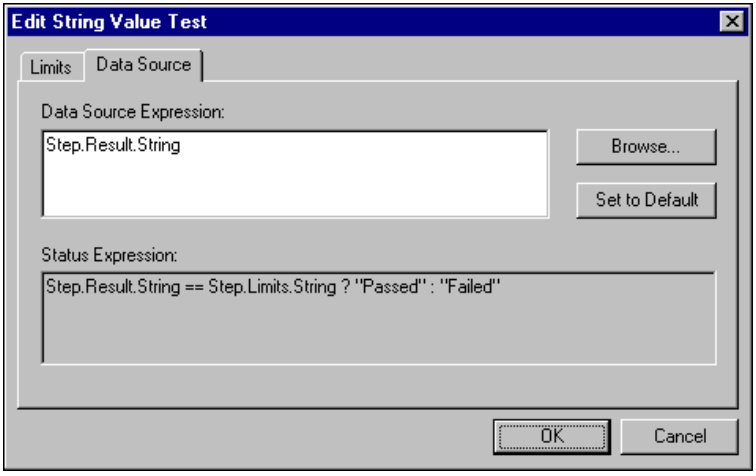

Figure 10-8. Data Source Tab on Edit String Value Test Dialog Box

The Data Source tab specifies a data source expression that TestStand evaluates to obtain the string it compares against the expected string.

Figure 10-9 shows the step properties for the String Limit Test step type.

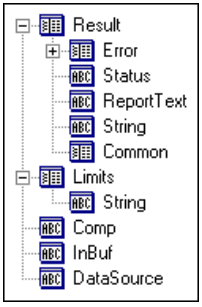

Figure 10-9. String Limit Test Step Properties

The String Value Test step type defines the following step properties in addition to the common custom properties.

- `Step.Result.String` specifies the string value. Usually, you set this value in the step module.
- `Step.Limits.String` specifies the expected string for the string comparison.
- `Step.Comp` specifies the type of comparison, such as `Ignore Case`.
- `Step.InBuf` specifies an arbitrary string that the C/CVI and LabVIEW module adapters automatically pass to the test in the `tTestData` structure or `TestData` cluster. This property exists to maintain compatibility with previous test executives. Usually, code modules that you develop for TestStand receive data as input parameters or access data as properties using the TestStand ActiveX API.
- `Step.DataSource` specifies a string expression that the step type uses to set the value of `Step.Result.String`. The default value of the expression is `Step.Result.String`, which has the effect of using the value that the code module sets. You can customize this expression if you do not want to set the value of `Step.Result.String` in the code module.

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

Step Types That Work With a Specific Module Adapter

This section describes step types that work with a specific module adapter.

Sequence Call

You use a Sequence Call step to call another sequence in the current sequence file or in another sequence file. A Sequence Call step always uses the Sequence Adapter.

You can use the Sequence Adapter with other step types such as Pass/Fail Test or Numeric Limit Test. Using a Sequence Call step is the same as using an Action step with the Sequence Adapter, except that the step type sets the step status to `Passed` rather than `Done` when the subsequence succeeds. If the sequence fails, TestStand sets the step status to `Failed`. A subsequence

fails when the status for a step in the subsequence is `Failed` and the Step Failure Causes Sequence Failure option for the step is enabled.

You specify the subsequence that the Sequence Call step executes by selecting the **Specify Module** item in the context menu for the step or clicking on the **Specify Module** button on the Step Properties dialog box.

Figure 10-10 shows the Specify Module dialog box for a Sequence Call step.

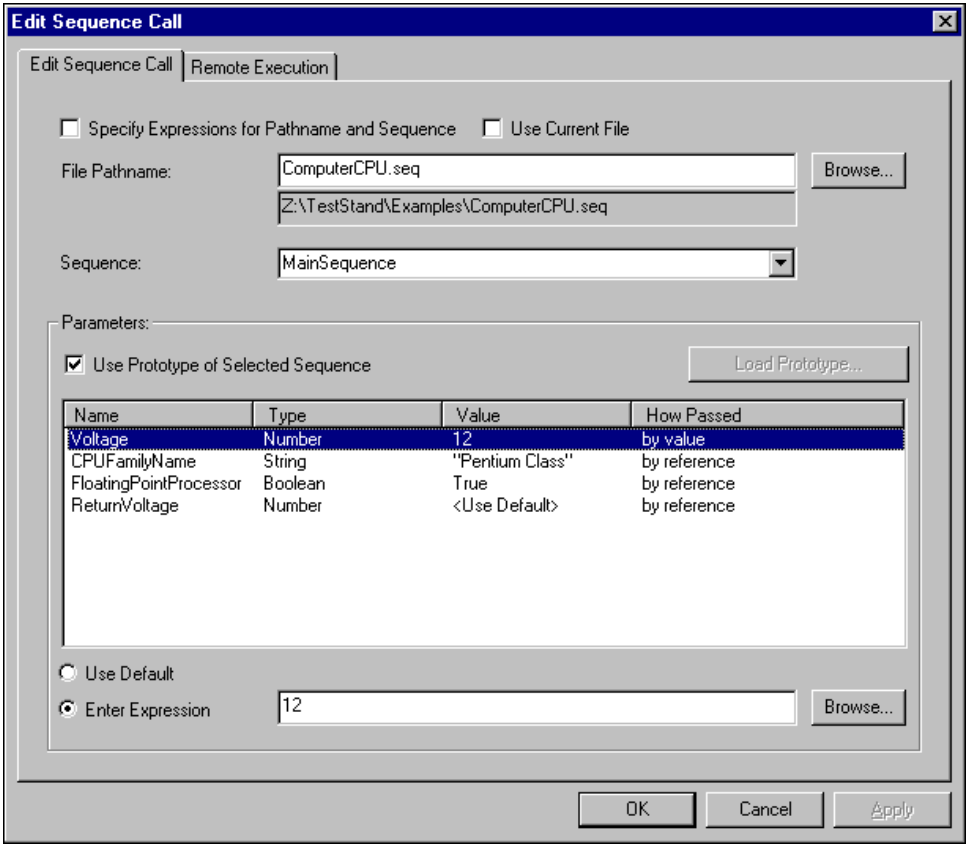

Figure 10-10. Specify Module Dialog Box for Sequence Call Step

You can specify the sequence and the sequence file using literal strings or expressions that TestStand evaluates at run-time.

Using the Parameters section of the dialog box, you can specify the values or expressions to pass for each parameter in the sequence call. For each

parameter, you can choose to use the default value for the parameter rather than specifying an explicit value. You can pass parameters to the sequence by value or by reference.

Refer to the *Sequence Adapter* section in Chapter 12, *Module Adapters*, for more information on using the Specify Module dialog box.

After the sequence call executes, the Sequence Adapter can set the step status. If the subsequence fails, the adapter sets the step status to `Failed`. If a run-time error occurs in the subsequence, the adapter sets the step status to `Error`. If the subsequence succeeds, the adapter does not set the step status. Instead, the Sequence Call step sets the step status to `Passed`.

The Sequence Call step type does not define any additional step properties other than the custom properties that are common to all steps.

Step Types That Do Not Use Module Adapters

This section describes step types that do not use module adapters. When you create an instance of one of these step types, you configure the step using a dialog box only. You do not write a code module.

Statement

You use Statement steps to execute expressions. For example, you can use a Statement step to increment the value of a local variable in the sequence file.

You can specify the expression for a Statement step by selecting the **Edit Expression** item in the context menu for the step or clicking on the **Edit Expression** button in the Step Properties dialog box.

Figure 10-11 shows the Edit Statement Step dialog box.

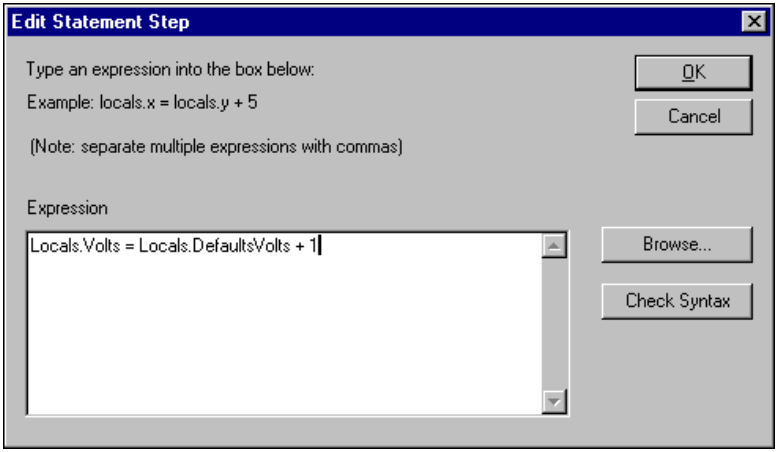

Figure 10-11. Edit Statement Step Dialog Box

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

The Statement step type does not define any additional step properties other than the custom properties that are common to all steps.

Message Popup

You use Message Popup steps to display messages to the operator and to receive response strings from the operator. For example, you can use a Message Popup step to warn the operator when a calibration routine fails.

You can specify the expression for the Message Popup step by selecting the **Edit Message Settings** item in the step context menu for the step or clicking on the **Edit Message Settings** button in the Step Properties dialog box.

Figure 10-12 shows the Configure Message Box Step dialog box.

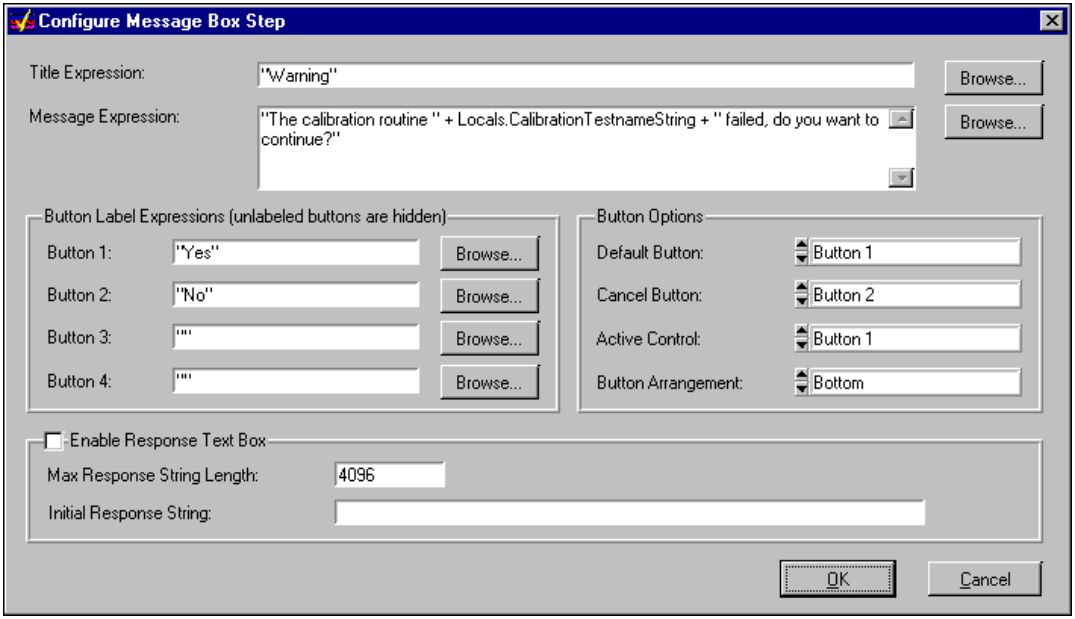

Figure 10-12. Configure Message Box Step Dialog Box

The Title Expression and Message Expression controls specify the text that the step displays in the message popup. In these two controls, you can specify literal strings or string expressions that TestStand evaluates at run-time. You can also customize expressions for each button label, and the arrangement of the buttons. If you do not specify a label for a button, the button does not appear on the popup. The Default Button selection ring control selects which button, if any, has <Enter> as its shortcut key. The Cancel Button selection ring control selects which button, if any, has <Esc> as its shortcut key. The Active Control selection ring control selects one of the four buttons or the input string as the initially active control.

You can also prompt the operator for a response by enabling the Enable Response Text Box option. You can specify the maximum response string length and an initial response string. If you do not want to specify a maximum response string length, specify `-1` in the Max Response String Length control.

After the operator closes the message popup, the step sets the `Step.Result.ButtonHit` step property to the number of the button that

the operator selects. The step copies the response string to `Step.Result.Response`.

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

Figure 10-13 shows the step properties for the Message Popup step type.

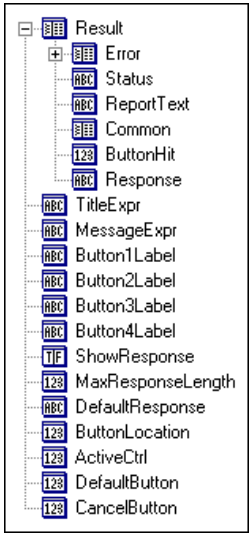

Figure 10-13. Message Popup Step Properties

The Message Popup step type defines the following step properties in addition to the common custom properties.

- `Step.Result.ButtonHit` contains the number of the button that the operator selects.
- `Step.Result.Response` contains the response text that the operator enters.
- `Step.TitleExpr` contains the expression for the string that appears as the title of the message popup.
- `Step.MessageExpr` contains the expression for the string that appears as the text message on the message popup.
- `Step.Button1Label`, `Button2Label`, `Button3Label`, and `Button4Label` specify the expression for the label text for each button.
- `Step.ShowResponse` enables the response text box control on the message popup.

- `Step.MaxResponseLength` specifies the maximum number of characters that the operator can enter in the response text box.
- `Step.DefaultResponse` contains the initial text string that the step displays in the response text box.
- `Step.ButtonLocation` specifies whether to display the buttons on the bottom or side of the message popup.
- `Step.ActiveCtrl` chooses one of the four buttons or the input string as the active control.
- `Step.DefaultButton` specifies which button, if any, has <Enter> as its shortcut key.
- `Step.CancelButton` specifies which button, if any, has <Esc> as its shortcut key.

Call Executable

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

You can specify the executable path, arguments, and options for the Call Executable step by selecting the **Configure Call Executable** item in the context menu for the step or the **Configure Call Executable** button on the Step Properties dialog box.

Figure 10-14 shows the Configure Call Executable dialog box.

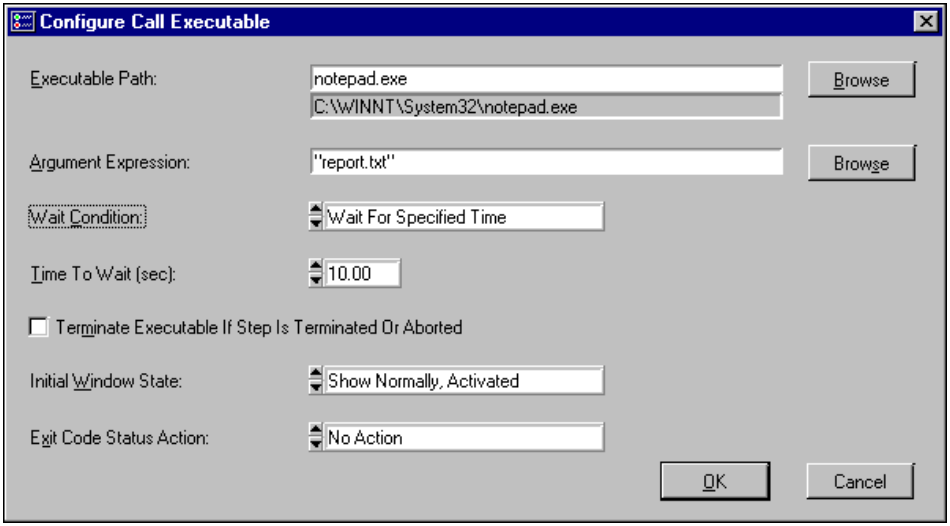

Figure 10-14. Configure Call Executable Dialog Box

The Configure Call Executable dialog box contains the following controls:

- **Executable Path**—Use this control to specify an absolute or relative pathname for the executable.
- **Argument Expression**—Use this control to specify an argument to pass to the executable. You can specify the argument as a literal string or as an expression that TestStand evaluates at run-time.
- **Wait Condition**—Use this control to specify whether the step waits for the executable to exit. The possible values are `No Wait`, `Wait for Exit`, and `Wait for Specified Time`. If you choose `Wait for Specified Time` and the executable process does not exit before the time limit you specify expires, the step type sets the `Step.Result.Error.Occurred` to indicate a run-time error.
- **Time to Wait**—Use this control to specify the time you want the step to wait for the executable to exit before it indicates a run-time error.
- **Terminate Executable If Step Is Terminated Or Aborted**—Use this control if you do not want the executable process to continue running when the operator terminates or aborts the execution in TestStand. This option applies only when the wait condition is Wait For Exit or Wait For Specified Time.

- **Initial Window State**—Use this control to specify whether the step launches the executable as a hidden, normal, minimized, or maximized application, and whether the application is active initially.
- **Exit Code Status Action**—Use this control to specify whether the step type sets the step status based on the exit code that the executable returns. You can choose to set the step status to `Failed` if the exit code is less than zero, greater than zero, equal to zero, or not equal to zero.

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

Figure 10-15 shows the step properties for the Call Executable step type.

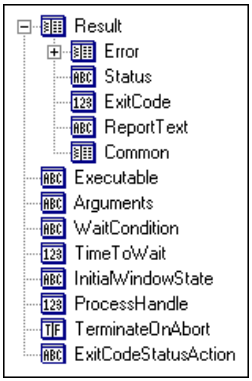

Figure 10-15. Message Popup Step Properties

The Message Popup step type defines the following step properties in addition to the common custom properties.

- `Step.Result.ExitCode` contains the exit code that the executable call returns.
- `Step.Executable` specifies the pathname of the executable to launch.

- `Step.Arguments` specifies the expression for the argument string that the step passes to the executable.
- `Step.WaitCondition` specifies whether the step waits for the executable to exit before completing.
- `Step.TimeToWait` specifies the number of seconds to wait for the executable to exit.
- `Step.ProcessHandle` contains the Windows process handle for the executable.
- `Step.InitialWindowState` specifies whether 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.
- `Step.ExitCodeStatusAction` specifies whether to set the step status using the exit code that the executable returns.

Limit Loader

You can use a Limit Loader step to update the limit properties of one or more steps in a sequence dynamically. For example, you might want to develop a common sequence that can test two different models of a cellular phone, where each model requires unique limit values for each step. If you use step properties to hold the limit values, you can include a Limit Loader step in the sequence to load the correct limit values into the step properties.

You usually insert the Limit Loader step in the Setup step group of a sequence. In this way, the Limit Loader step initializes the limit values before the steps in the Main step group execute.

The source of the limit values can be a tab-delimited text file (`.txt`), a comma-delimited text file (`.csv`), or an Excel file (`.xls`). The limit data is in a table format where the row names are step names and the column headings are the names of step properties that begin with `Limits`. Starting and ending data markers designate the bounds of the table. A limit file can contain more than one block of data. The following is an example of a tab-delimited limits file with one data block.

```
Start Marker
                  Limits.Low Limits.High Limits.String
Voltage at Pin A  9.000000   11.000000
Voltage at Pin B  8.500000   9.5000000
Self Test Output                        "SYSTEM OK"
End Marker
```

Figure 10-16 shows an example sequence file that might use the limits file.

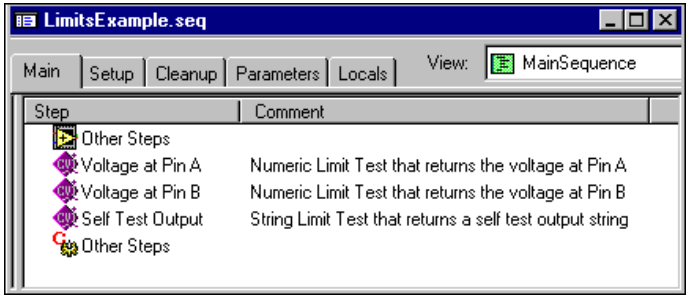

Figure 10-16. Example Sequence File with Limit Steps

You can specify the pathname and layout of the limits file in a Limit Loader step by selecting the **Select Limits File** item in the context menu for the step or clicking on the **Select Limits File** button on the Step Properties dialog box.

Figure 10-17 shows the Limits File tab of the Edit Limit Loader Step dialog box.

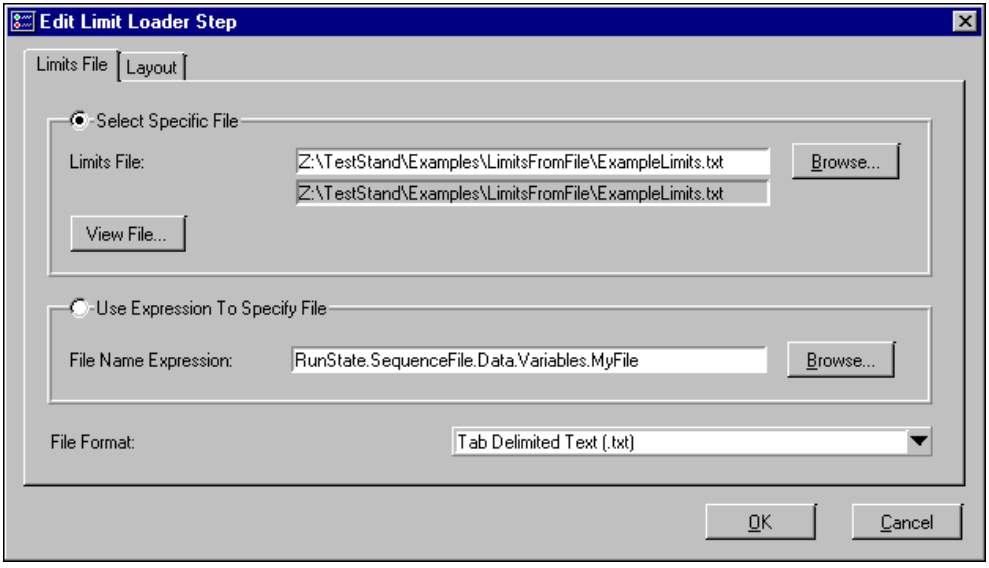

Figure 10-17. Limits File Tab on Edit Limit Loader Step Dialog Box

On the Limits File tab, you can select a specific limits file, or you can specify a string expression that TestStand evaluates at run-time for the limits file pathname. You must also select a file format for the limits file. Valid formats are tab-delimited text (`.txt`), comma-delimited text (`.csv`), and Excel file (`.xls`).

The Layout tab specifies the organization of the data in the limits file. Figure 10-18 shows the Layout tab of the Edit Limit Loader Step dialog box.

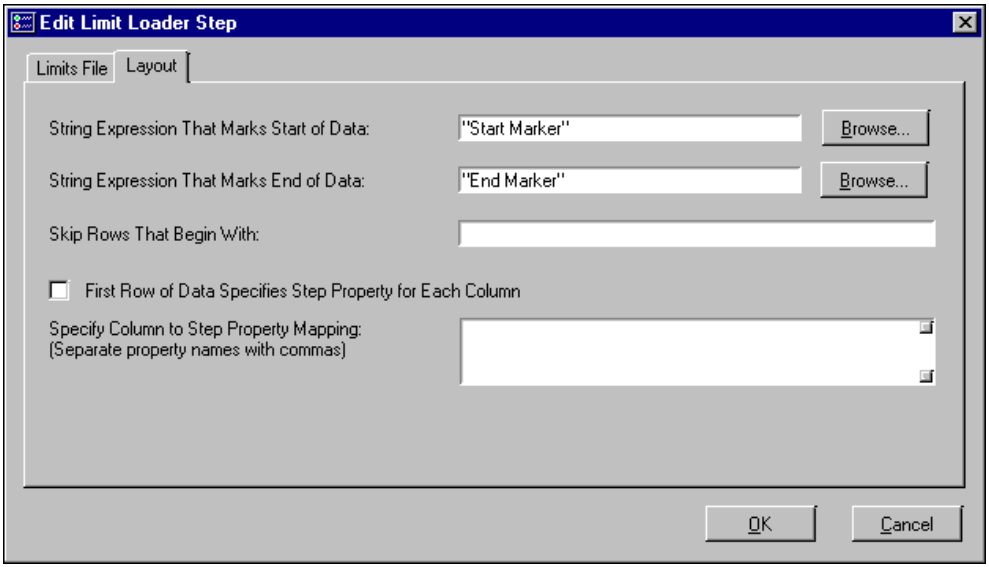

Figure 10-18. Layout Tab on Edit Limit Loader Step Dialog Box

The String Expression that Marks Start of Data control specifies the string that designates the beginning of a block of limit data. The String Expression that Marks End of Data control specifies the string that designates the end of a block of limit data. You can specify literal strings for the beginning and ending markers, or you can specify string expressions that TestStand evaluates at run-time. The marker strings must appear in the first column of the file. If you specify an empty expression (`""`) for the start and end markers, the step type assumes that the file contains a single block of limit data.

A Limit Loader step ignores all rows that begin with the string you specify in the Skip Rows that Begin With control. This feature is useful if the limits file includes comment lines.

Disable the First Row of Data Specifies Step Property for Each Column option if you do not want to include the step property names for each column as the first row of each data block in the limits file. If you disable this option, you must use the Specify Column to Step Property Mapping text box to specify the list of property names. Separate the property names with commas, as in the following example.

```
Limits.Low, Limits.High, Limits.String
```

Figure 10-19 shows the step properties for the Limit Loader step type.

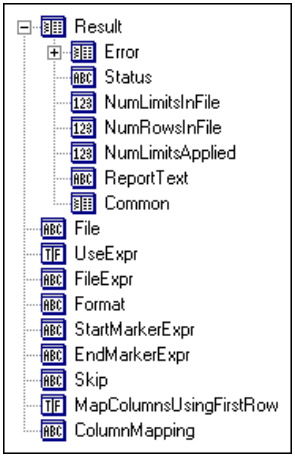

Figure 10-19. Limit Loader Step Properties

The Limit Loader step type defines the following step properties in addition to the common custom properties.

- `Step.Result.NumLimitsInFile` indicates the total number of limit values in the data block that the step loaded from the file.
- `Step.Result.NumRowsInFile` indicates the number of limit value rows in the data block that the step loaded from the file.
- `Step.Result.NumLimitsApplied` contains the total number of limit values that the step assigned to limit properties in the sequence. If this number is less than `Step.Result.NumLimitsInFile`, the step was unable to find steps or properties in the sequence for all the step names and properties names in the file.

- `Step.File` specifies a literal pathname for the limit file.
- `Step.FileExpr` specifies a pathname expression that TestStand evaluates at run-time.
- `Step.UseExpr` specifies whether to use `Step.File` or `Step.FileExpr` for the pathname of the limits file.
- `Step.Format` specifies the layout of the file. The possible values are `Tab`, `Comma`, or `Excel`.
- `Step.StartMarkerExpr` specifies the expression for the starting marker.
- `Step.EndMarkerExpr` specifies the expression for the ending marker.
- `Step.Skip` specifies the string that, when it appears at the beginning of a row, causes the Limit Loader step type to ignore the row.
- `Step.MapColumnsUsingFirstRow` specifies whether the first row of each data block in the limit file contains the names of the step properties into which the Limit Loader step loads the limit values.
- `Step.ColumnMapping` specifies the names of the properties into which the Limit Loader step loads the limit values if `Step.MapColumnsUsingFirstRow` is `False`.

Import/Export Limits Command in the Tools Menu

When you edit a sequence file, you can use the **Import/Export Limits** command in the **Tools** menu to import limit values into a sequence from a limits file or export limits from a sequence to a limits file.

The **Import/Export Limits** command displays the Import/Exports Sequence Limits dialog box, as shown in Figure 10-20.

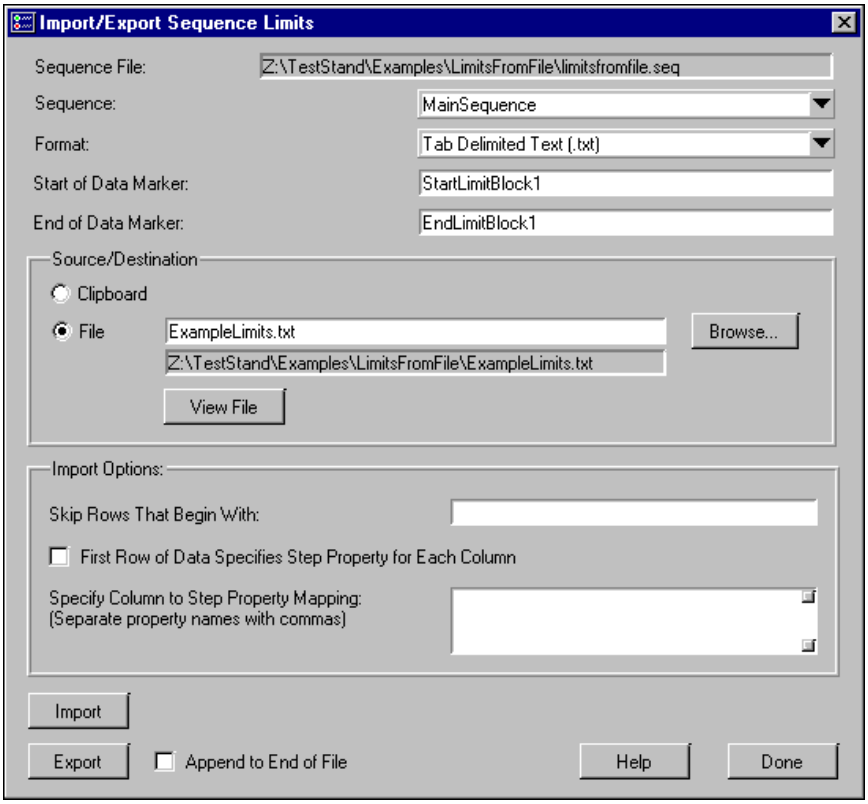

Figure 10-20. Import/Exports Sequence Limits Dialog Box

The Import/Export Sequence Limits dialog box contains the following options:

- **Sequence File**—This indicator displays the name of the sequence file that the current invocation of the **Import/Export Limits** command applies to.
- **Sequence**—Use this control to select the sequence into which to import limit values or from which to export limit values.

- **Format**—Use this control to specify the file format for the limits file. The file format can be tab-delimited text (`.txt`), comma-delimited text (`.csv`), and Excel file (`.xls`). The limit data is in table format where the row names are step names and the column headings are the names of step properties that begin with `Limit`. The following is an example tab-delimited limits file with one data block.

```
StartLimitBlock
                  Limits.Low Limits.High Limits.String
Voltage at Pin A  9.000000   11.000000
Voltage at Pin B  8.500000   9.5000000
Self Test Output                         "SYSTEM OK"
EndLimitBlock
```

- **Start of Data Marker**—Use this control to specify a string that designates the beginning of a block of limit data. (The marker string must appear at the beginning of a row.)
- **End of Data Marker**—This marker specifies the string that designates the end of a block of limit data. (The marker string must appear at the beginning of a row.)
- **Source/Destination**—This section of the dialog box specifies the external location from which you import, or into which you export, the limit values in the sequence. You can specify the system clipboard or a specific file.
- **Skip Rows That Begin With**—This option ignores all rows that begin with the string that you specify in the Skip Rows that Begin With control. This feature is useful if the limits file includes comment lines.
- **First Row of Data Specifies Step Property for Each Column**—Disable this option if you do not want to include the step property names for each column as the first row of each data block in the limits file. If you disable this option, you must use the Specify Column to Step Property Mapping text box to specify the list of property names. Separate the property names with commas, as in the following.

  ```
  Limits.Low, Limits.High, Limits.String
  ```

- **Import**—Click on this button to import limit values from a file or the system clipboard into a sequence. The source must contain a block of limit values starting and ending with the data markers you specify. The **Import** command displays the number of limit values it successfully imports, and lists any steps or step property names that it cannot find in the destination sequence.

- **Export**—Click on this button to export limit values from a sequence to a file or the system clipboard. If the destination is a file that already exists and the Append to End of File option is disabled, a dialog box appears prompting you to overwrite the file. The **Export** command writes a block of limit data with the starting and ending markers you specify to the file or clipboard.
- **Append to End of File**—Disable this option if you want to overwrite the file when you export a block of limit data. Enable this option if you want to append a block of data to a file that already exists.

Goto

You use Goto steps to set the next step that the TestStand engine executes. You usually use a Label Step as the target of a Goto step. This allows you to rearrange or delete other steps in a sequence without having to change the specification of targets in Goto steps.

You can specify the Goto step target by selecting the **Edit Destination** item from the step context menu or clicking on the **Edit Destination** button on the Step Properties dialog box. Figure 10-21 shows the Edit Goto Step dialog box.

Figure 10-21. Edit Goto Step Dialog Box

The Destination control contains a list of all steps in the step group. The Destination control lists two additional targets: <Cleanup> allows you to jump to the Cleanup step group, and <End> allows you to jump directly to the end of the current step group.

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

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

Label

You usually use a Label Step as the target for a Goto step. This allows you 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. After a Label step executes, the TestStand engine sets the step status to `Done` or `Error`.

The Label step type does not define any additional step properties other than the custom properties common to all steps.

11

User Management

This chapter describes TestStand user management, the User Manager window, and how you can add users and manage user privileges.

The TestStand engine maintains a list of users, their login names and passwords, and their privileges. This capability of the TestStand engine is called the *user manager*. The sequence editor and operator interfaces limit their available functionality depending on the privilege settings that the user manager stores for the user that is currently logged in.

When you launch the sequence editor or any operator interfaces that come with TestStand, they each display the Login dialog box by calling the `LoginLogout` front-end callback sequence. The `LoginLogout` sequence calls the `DisplayLoginDialog` method of the `Engine` class, which displays the actual dialog box.

The User Manager tab of the Station Options dialog box specifies whether TestStand enforces user privileges and specifies the location of the user manager configuration file. Refer to the *Configure Menu* section in Chapter 4, *Sequence Editor Menu Bar*, for more information on these options.

Note ***The TestStand User Manager is designed to help you implement policies and procedures concerning the use of your test station. It is not a security system and it does not inhibit or control the operating system or third-party applications. You must use the system-level security features provided by your operating system to secure your test station computer against malicious use.***

User Manager Window

In the TestStand sequence editor, you use the User Manager window to view and edit the user list and the privileges of each user. You can open the User Manager window by selecting **View»User Manager**. You use the View ring at the top right of the User Manager window to choose whether to display the list of users or the list of types that TestStand uses to store user privileges.

Users View

You can display the list of users by selecting Users from the View ring. You can use this view to add new users or to modify the privileges and other properties of existing users. Figure 11-1 shows the Users view in the User Manager window.

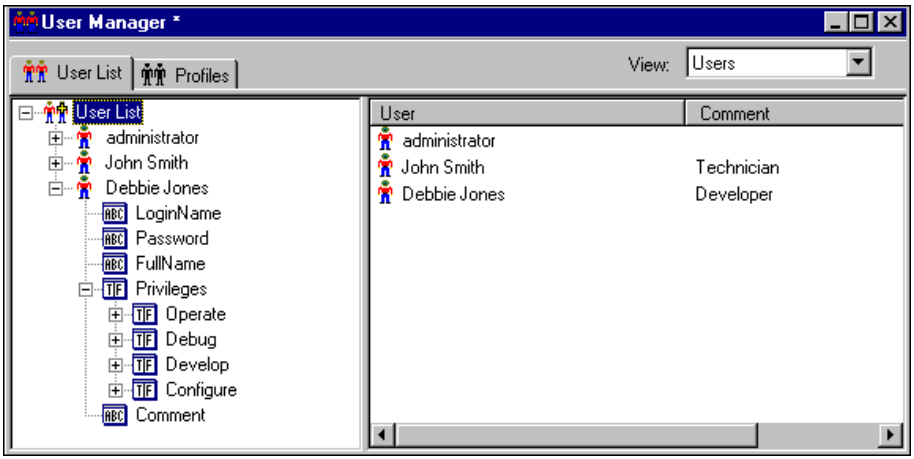

Figure 11-1. Users View in the User Manager Window

The Users view has two tabs: User List and Profiles. The User List tab contains a list of current users. Each entry contains properties that define the login name, the login password, and the TestStand privileges. TestStand stores these properties in User containers, which have the `User` standard data type.

The Profiles tab contains a list of profiles that you can apply when you create new users. A profile defines a set of values for the properties in the `User` data type. When you create a new user, you can initialize the values for a new user from a profile. If you make changes to the values in a profile, your changes do not affect the privileges for users already in the user list. TestStand defines four default profiles: *operator*, *technician*, *developer*, and *administrator*.

User List Tab

The User List tab contains two panes. The left pane is a tree view that allows you to browse the custom properties for each user. The right pane is a list view that displays the contents of the node you select in the tree view.

The columns in the list view vary according to whether the list view is displaying users or user properties. When the list view displays users, the columns appear as in Figure 11-1.

Figure 11-2 shows the columns that appear when the list view displays user properties.

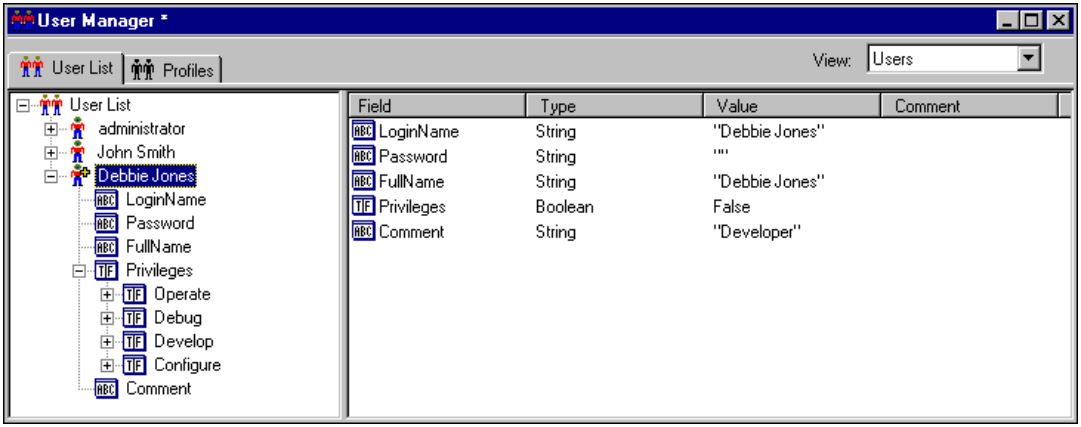

Figure 11-2. User List Tab for Users View

User List Context Menu

You can display a context menu by right-clicking on the tree view or list view. The items in the context menu vary depending on the whether you right-click on a step, a step property, the background area of the tree view, or the background area of the list view. The User List tab context menu contains the common editing and navigation commands, and the following additional commands.

Insert User

You can use the **Insert User** command to add a new user to the user list. Figure 11-3 shows the New User dialog box.

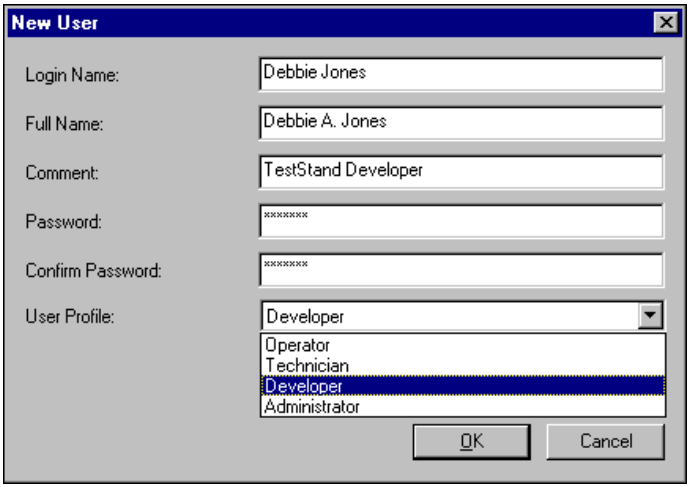

Figure 11-3. Insert New User Dialog Box

The Login Name and the Password controls specify a case-sensitive login name and password. You can use the Full Name and Comment controls to add additional information about the user. The User Profile ring control selects a profile, which defines an initial set of privilege settings to give the new user.

Edit

You can use the **Edit** command to edit an existing user in the user list. Figure 11-4 shows the Edit User dialog box.

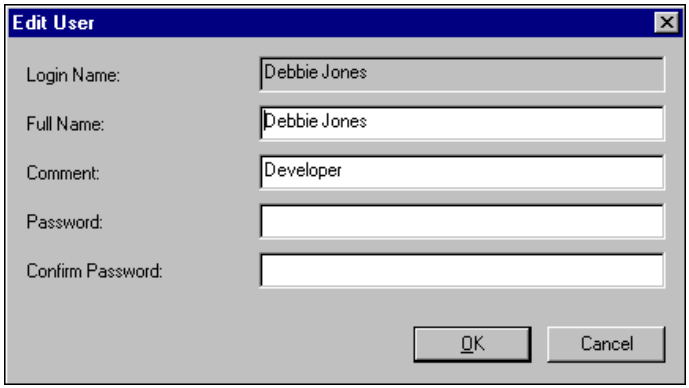

Figure 11-4. Edit User Dialog Box

Edit User Type

The **Edit User Type** command switches from the Users view to the Types view in the User Manager window and displays the `User` standard data type.

Profiles Tab

The Profiles tab contains two panes. The left pane is a tree view that allows you to browse the custom property values for each profile. The right pane is a list view that displays the contents of the item you select in the tree view. The columns in the list view vary according to whether the list view is currently displaying profiles or profile properties.

Figure 11-5 shows the Profiles tab in the Users view.

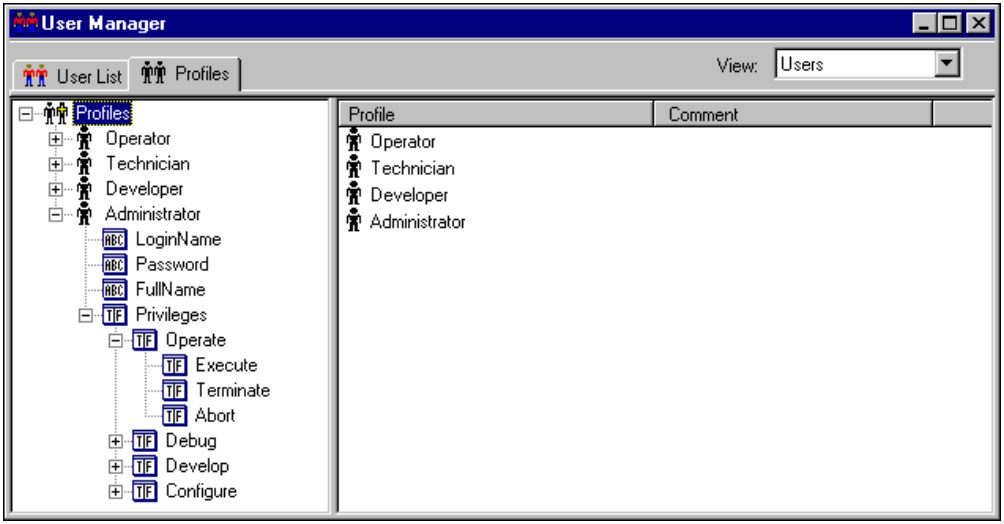

Figure 11-5. Profile Tab in the Users View

The tree view in Figure 11-5 shows the set of privileges that the each of the profiles define values for. The User standard data type defines this set of privileges. The privileges are grouped in categories such as Operate and Debug. Each profile defines values for each of the privilege settings. Each user also has these privilege properties. When you create a new user, TestStand copies the privilege setting values from the profile you choose to the new user.

Profiles Tab Context Menu

You can display a context menu by right-clicking on the tree view or list view. The items in the context menu vary depending on whether you right-click on a step, a step property, the background area of the tree view, or the background area of the list view. The Profiles tab context menu contains the common editing and navigation commands, and the following additional commands.

Insert Profile

You can use the **Insert Profile** command to add a new profile to the profile list.

Edit User Type

The **Edit User Type** command switches from the Types view to the Users view in the User Manager window and displays the `User` standard data type.

Types View

You can display a list of the types that the User Manager uses by selecting Types from the View ring. You can use this view to add new properties to the `User` data type, or create your own custom data types to add to the `User` data type. Figure 11-6 shows the Types view for the User Manager Window. The Types view has two tabs, Standard Data Types and Custom Data Types.

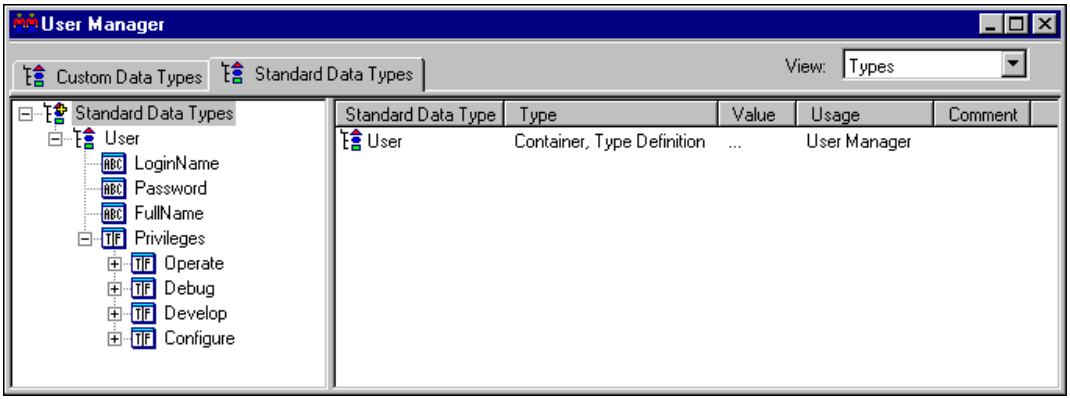

Figure 11-6. Types View in the User Manager Window

User Standard Data Types

The Standard Data Types tab contains the standard data types that the User Manager uses. TestStand stores the properties for each user in a User container, which has the `User` standard data type.

Figure 11-7 shows the tree view of the `User` standard data type.

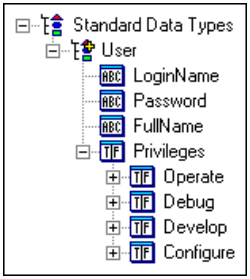

Figure 11-7. User Standard Data Type

Table 11-1 lists the properties in the `User` data type and the privileges that TestStand grants a user when the property value is `True`. Subproperties appear in the table indented under the properties they belong to. For example, you reference the `Execute` subproperty as `User.Privileges.Operate.Execute`.

Table 11-1. Description of Subproperties in User Data Type

Subproperty	Description
`LoginName`	User login name.
`Password`	User login password.
`FullName`	Descriptive user name.
`Privileges`	User has all TestStand privileges. When `True`, TestStand ignores all specific privilege settings in all property groups.
`Operate`	User can perform all `Operate` privileges. When `True`, TestStand ignores all specific privilege settings in this property group.
`Execute`	User can initiate an execution.
`Terminate`	User can terminate an execution.

Table 11-1. Description of Subproperties in User Data Type (Continued)

Subproperty	Description
`Abort`	User can abort an execution.
`Debug`	User can perform all `Debug` privileges. When `True`, TestStand ignores all privilege settings in this property group.
`ControlExecFlow`	User can control the flow of execution by setting breakpoints, single-stepping, and using the **Set Next Step** and **Run Mode** commands.
`SinglePass`	User can use the Single Pass execution entry point.
`RunAnySequence`	User can run individual sequence without using execution entry points.
`RunSelectedTests`	User can run selected tests from a sequence using the **Run Selected Steps** command.
`LoopSelectedTests`	User can run selected tests from a sequence in a loop using the **Loop on Selected Steps** command.
`EditStationGlobals`	User can create and modify globals in the Station Globals Window.
`Develop`	User can perform all `Develop` privileges. When `True`, TestStand ignores all privilege settings in this property group.
`EditSequenceFiles`	User can edit sequence files, sequences, and steps.
`SaveSequenceFiles`	User can save sequence files.
`Configure`	User can perform all `Configure` privileges. When `True`, TestStand ignores all privilege settings in this property group.
`EditTypes`	User can create and modify standard data types, custom data types, and step types.

Table 11-1. Description of Subproperties in User Data Type (Continued)

Subproperty	Description
`ConfigEngine`	User can configure the engine, which includes: • Customizing the **Tools** menu. • Modifying engine files in the Edit Paths dialog box. • Modifying settings on the Search Directories dialog box. • Modifying settings on all tabs of the Station Options dialog box except Preferences and User Manager tabs. • Modifying the Prompt to Find Files option on the Preferences tab of the Station Options dialog box.
`ConfigAdapter`	User can configure adapters in the Adapter Configuration dialog box.
`ConfigApp`	User can modify the settings on the Preferences tab of the Station Options dialog box, except for the Prompt to Find Files option.
`ConfigReport`	User can modify the settings in the Report Options dialog box.
`ConfigDatabase`	User can modify the settings in the Database Options dialog box.
`EditUsers`	User can add and modify users in the User Manager Window and the options on the User Manager tab of the Station Options dialog box.

Notice that each privilege group has a Boolean value. When the Boolean value for a privilege group is `True`, it overrides the values of each privilege in the group. When the Boolean value for a group is `False`, the value of each privilege in the group applies.

Adding New Properties and Privileges to the User Data Type

You can add new subproperties to the `User` data type. For example, you might want to add an ID string property for each user, or you might want to add a `ConfigHardware` Boolean property in the

`User.Privilege.Configure` group to specify whether a user has the privilege to configure special hardware on the station.

You can use the Custom Data Types tab to define your own data types, which you can then add to the `User` standard data type. Refer to the *Using Data Types* section in Chapter 9, *Types*, for more information on data types and editing data types.

Verifying User Privileges

This section discusses how you can verify that a user has a specific privilege.

Accessing Privilege Settings for the Current User

If you want to verify in an expression that the current user has a specific privilege, you can call the `CurrentUserHasPrivilege` expression function. If you want to verify it in a code module, you can call the `CurrentUserHasPrivilege` method of the Engine class in the TestStand ActiveX API.

When you call the `CurrentUserHasPrivilege` expression function, you must specify the property name of the privilege as a string argument. The current user has a privilege if the property is `True` or if any parent Boolean property is `True`. For example, a user has the privilege to terminate an execution if either the value of the `User.Privileges.Configure.Terminate` property is `True` or the value of the `User.Privileges.Configure` group is `True`. The `CurrentUserHasPrivilege` function returns `True` if the current user has the privilege or privilege checking is disabled.

You can pass any subset of the property name tree structure to the `CurrentUserHasPrivilege` function. For example, you can use either of the following two expressions to determine whether the current user has the privilege to terminate an execution.

```
CurrentUserHasPrivilege("Terminate")
CurrentUserHasPrivilege("Configure.Terminate")
```

You can pass `"*"` as the string argument to `CurrentUserHasPrivilege` to determine if a user is currently logged in. Refer to Chapter 8, *Sequence Context and Expressions*, for more information on using expressions.

The `CurrentUserHasPrivilege` method behaves identically to the expression function, except that it takes additional parameters. Refer to the *TestStand ActiveX API Reference* online help for more information.

Accessing Privilege Settings for Any User

The TestStand ActiveX API has methods you can use to access the privileges of any user. You can use the `GetUser` method of the `Engine` class to return a `User` object. You can then use the `HasPrivilege` method in the `User` class to inspect the value of a specific privilege. The `HasPrivilege` method behaves identically to the `CurrentUserHasPrivilege` expression function. Refer to the *TestStand ActiveX API Reference* online help for more information.

12

Module Adapters

This chapter describes the module adapters that TestStand includes.

Overview

The TestStand engine uses a module adapter to invoke the code in a step module. Each module adapter supports one or more specific types of code modules. The different types of code modules include TestStand sequences, LabVIEW VIs, ActiveX Automation Objects, C functions in DLLs, and C functions in source, object, or library modules that you create in LabWindows/CVI or other compilers. A module adapter knows how to load and call a code module, how to pass parameters to a code module, and how to return values and status from a code module.

When you edit a step that uses a module adapter, TestStand relies on the adapter to display a dialog box in which you can specify the code module for the step along with any parameters to pass when invoking the code module. This dialog box is called the Specify Module dialog box. The actual title of the dialog box is different for the different adapters. TestStand stores the name and location of the code module, the parameter list, and any additional options as properties of the step. TestStand hides most of these adapter-specific step properties.

You can display the Specify Module dialog box for a step that calls a code module by selecting the **Specify Module** command from the step context menu or clicking on the **Specify Module** button in the Step Properties dialog box.

If the module adapter is specific to an ADE, the adapter knows how to open the ADE, how to create source code for a new code module in the ADE, and how to display the source for an existing code module in the ADE. If source code is available for the code module, the adapter also supports stepping into the code in the ADE when you execute the step from the TestStand sequence editor.

TestStand currently provides the following module adapters:

- **DLL Flexible Prototype Adapter**—Allows you to call C functions in a DLL with a variety of parameter types.
- **LabVIEW Standard Prototype Adapter**—Allows you to call any LabVIEW VI that has the TestStand standard G parameter list.
- **C/CVI Standard Prototype Adapter**—Allows you to call any C function that has the TestStand standard C parameter list. The function can be in an object file, library file, or DLL. It also can be in a source file when you are using the LabWindows/CVI development environment and the source file is in the LabWindows/CVI project.
- **Sequence Adapter**—Allows you to call subsequences with parameters.
- **ActiveX Automation Adapter**—Allows you to call methods and access the properties of an ActiveX object.

Configuring Adapters

Some of the module adapters are configurable. You can configure a module adapter by selecting the **Adapters** command in the **Configure** menu on the TestStand main menu.

Figure 12-1 shows the Adapter Configuration dialog box.

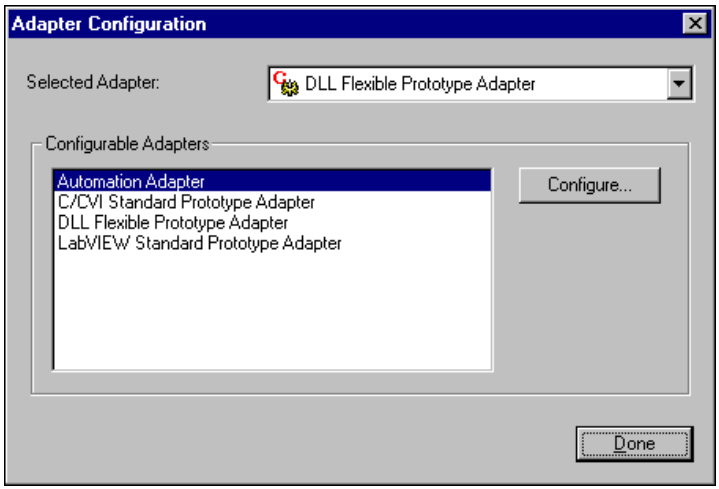

Figure 12-1. Adapter Configuration Dialog Box

The Selected Adapter ring control specifies the module adapter that a new step you create uses. The selected adapter applies only to step types that can use any module adapter, such as the Action, Numeric Limit Test, String Limit Test, and Pass/Fail Test step types. Refer to the discussion on the **Insert Step** command in the *Step Group Context Menu* section of Chapter 5, *Sequence Files*, for more information on how TestStand uses the selected adapter when you insert a step.

To configure an adapter, select an adapter from the Configurable Adapters list and click on the **Configure** button. The **Configure** button displays an adapter-specific dialog box for configuring the adapter. Refer to the adapter-specific sections in this chapter for information on the specific configuration options for each adapter.

Source Code Templates

With some module adapters, you can use a source code template to generate the source code shell for a step module. The template files are different for each step type and each module adapter. Multiple source code templates can be available for a particular adapter/step type combination. Currently, only the LabVIEW, the DLL Flexible Prototype, and the C/CVI Standard Prototype Adapters support source code templates.

For each module adapter that supports source code templates, the Specify Module dialog box displays a command button for creating source code for the step from a template. If more than one template is available for the adapter/step type combination you use to create the step, the adapter prompts you to select from a list of the templates as shown in Figure 12-2.

If only one template is available, the adapter uses that template automatically.

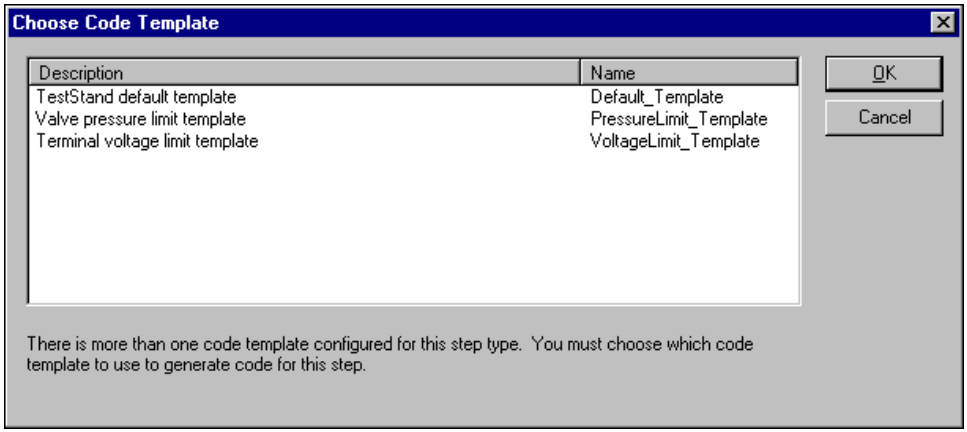

Figure 12-2. Choose Code Template Dialog Box

TestStand ships with default templates for each of the built-in step types. You can create additional templates for built-in step types. When you create a new step type, you can create one or more source code templates for it. Refer to the *Using Step Types* section in Chapter 9, *Types*, for more information on creating source templates for step types.

DLL Flexible Prototype Adapter

The DLL Flexible Prototype Adapter allows you to call C functions in a DLL with a variety of parameter types. You can create the DLL code module with LabWindows/CVI or any other ADE that creates a C DLL.

Configuring the DLL Adapter

The DLL Adapter Configuration dialog box contain a single option, Show Function Parameters in Function Description. When enabled, the description for a step lists the function and its parameters. When disabled, the description lists the function and the DLL name.

Specifying a DLL Adapter Module

The Specify Module dialog box for the DLL Flexible Prototype Adapter is called the Edit DLL Call dialog box. The Edit DLL Call dialog box contains a Module tab and a Source Code tab. The Module tab specifies the code module that the adapter executes for the step and parameter information for the module. The Source Code tab contains additional information that TestStand requires if you want to create and edit a code module in another application program.

Module Tab

Figure 12-3 shows the Module tab of the Edit DLL Call dialog box.

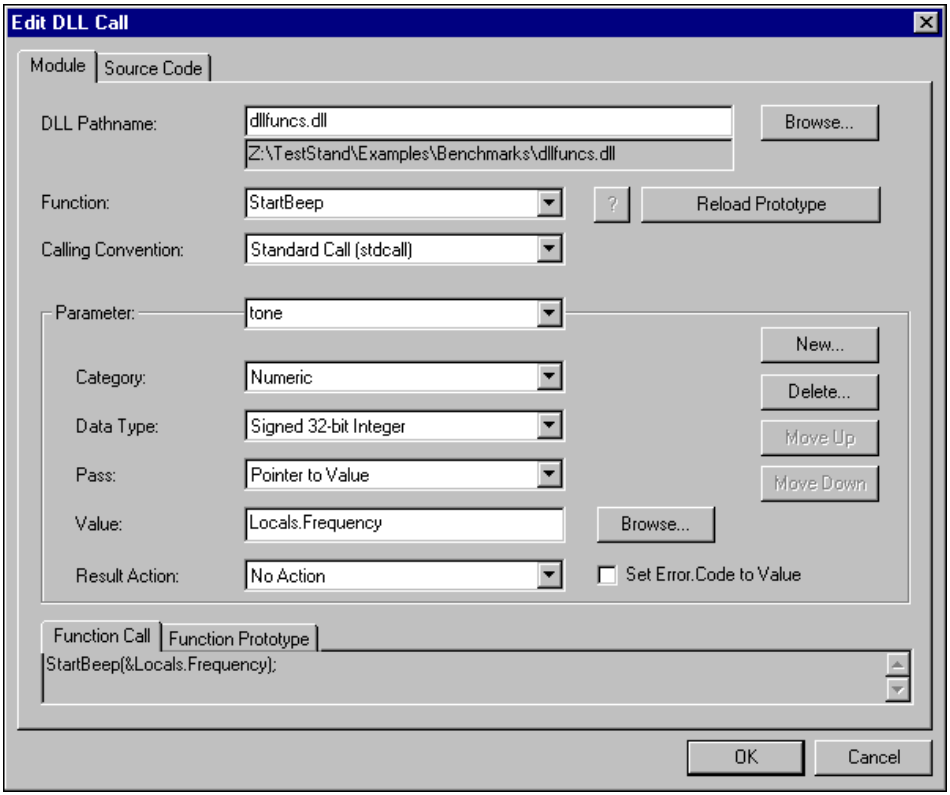

Figure 12-3. Specify Module Dialog Box for DLL Flexible Prototype Adapter—Module Tab

- **DLL Pathname**—Specifies the DLL file that contains the function the step calls. You can specify an absolute or relative pathname for the DLL file. Relative pathnames are relative to the TestStand search directory paths.

  You can customize the TestStand search directory paths with the **Search Directories** command in the **Configure** menu of the sequence editor menu bar.

- **Function**—Selects the function in the DLL that the step calls. If the DLL file contains a type library, the adapter automatically populates the function ring with all the function names in the type library. Otherwise, the adapter reads the DLL file and finds the names of all functions that the DLL exports. If a DLL type library contains links to a help file for a function, you can click on the **?** button to display the help.

- **Calling Convention**—Use this control to specify the calling convention of the function.

- **Parameter**—Use this section of the dialog box to specify the data type of the return value and of each parameter.

  For each parameter you also specify the value or expression to pass. If a DLL file contains a type library, the adapter queries the type library for the parameter list information and displays it in the parameter section automatically when you select a new function in the Function ring control. At any time, you can request the adapter to query the type library for the currently selected function by clicking on the **Reload Prototype** button. If the DLL does not have type library information, you can enter parameter information manually.

  The Parameter ring control lists a symbolic name for each parameter and a special entry for the return value. When you select a parameter in the ring control, the type of controls in the Parameter section change. You can insert and remove parameters by clicking on the **New** and **Delete** buttons. To rearrange the parameter order, select the parameter you want to move and select the **Move Up** or **Move Down** button.

- **Category**—Use this control to specify a group of data types to list in the DataType control. The categories include Numeric, String, Array, and Object. The Data Type control specifies the data type of the function parameter.

Numeric Parameters

Table 12-1 shows the Numeric category data types.

Table 12-1. TestStand Numeric Data Types

Numeric Data Type Setting	Equivalent C Data Type
Unsigned 8-bit Integer	`unsigned char`
Signed 16-bit Integer	`short`
Unsigned 16-bit Integer	`unsigned short`
Signed 32-bit Integer	`long`
Unsigned 32-bit Integer	`unsigned long`
32-bit Real Number	`float`
64-bit Real Number	`double`

When you select the Numeric category for a parameter, the adapter displays the Pass control. This control specifies whether TestStand passes the value of the argument you specify in the Value control, or passes a pointer to the argument.

If you choose to pass a pointer, the argument you specify in the Value control must be the name of a station global variable, sequence file global variable, sequence parameter, sequence local variable, or step property. When you select the Numeric category for the return value, you can leave the Value control empty or specify the name of a station global variable, sequence file global variable, sequence parameter, sequence local variable, or step property.

The adapter displays the Result Action ring control and the Set `Error.Code` to Value checkbox for return values and parameters you pass by pointer. Depending on the settings of these controls, TestStand can set the `Error.Occurred` and `Error.Code` properties of the step automatically based on the value in the numeric argument when the function returns.

You can use the Result Action control to configure TestStand to set the `Error.Occurred` property to `True` when the return value or parameter value after the call is greater than zero, less than zero, equal to zero, or not equal to zero. You can use the Set `Error.Code` to Value checkbox to request TestStand to assign the output value of the argument to the `Error.Code` property.

String Parameters

In general, when using string parameters, use one of the buffer types if you want the DLL function to be able to change the contents of the argument in TestStand. Use the C String or Unicode String type if the DLL function does not modify the argument.

Table 12-2 shows the String category data types.

Table 12-2. TestStand String Data Types

String Data Type Setting	Equivalent C Data Type
C String	`const char *`
C String Buffer	`char[]`
Unicode String	`const wchar_t *` or `const unsigned short *`
Unicode String Buffer	`wchar_t[]` or `unsigned short[]`

You can pass a string literal, a TestStand string property, or an expression that evaluates to a string as the value of a string parameter.

If you specify one of the string buffer types, the adapter copies the contents of the string argument and a trailing zero element into a temporary buffer before calling the DLL function. You specify the minimum size of the temporary buffer in the Number of Elements control. If the string value is longer than the buffer size you specify, the adapter sizes the temporary buffer so that it is large enough to hold contents of the string argument and the trailing zero element. After the DLL function returns, TestStand resets the value of the string argument from the contents of the temporary buffer.

If you specify the C String or Unicode String type, the adapter passes the address of the actual string directly to the function without copying it to a buffer. The code module must not change the contents of the string.

Array Parameters

The Array category contains the same data types as the numeric category. You can specify an array that contains elements of any numeric type. TestStand reformats the contents of the numeric array argument into a temporary array containing elements that have the data type you select.

You use the Number of Elements control to specify the number of elements in the temporary array. If the array argument has fewer elements than the temporary array, the adapter fills out the remaining elements in the temporary array with zeroes. If the array argument has more elements than the temporary array, TestStand reformats as many elements as can fit into the temporary array.

If you want the number of elements in the temporary array to always match the number of elements in the array argument, specify a negative value in the Number of Elements control. This is equivalent to the following expression:

`GetNumElements (`*`arrayArgument`*`)`

If you specify a zero value in the Number of Elements control, TestStand passes the address of a temporary array with no elements to the DLL function.

When the DLL function returns, TestStand reformats the contents of the temporary array into the array argument. If the array argument has fewer elements than the temporary array, TestStand reformats as many elements as can fit into the array argument. If the array argument has more elements than the temporary array, TestStand reformats all the elements of the temporary array into the array argument and leaves the remaining elements of the array argument alone.

Object Parameters

The Object category includes the ActiveX Automation `IDispatch` Pointer, ActiveX Automation `IUnknown` Pointer, and LabWindows/CVI ActiveX Automation Handle data types. You can use these types to pass a reference to a built-in or custom TestStand object to the DLL function, or pass the value of an ActiveX reference property to the DLL function.

If you specify an ActiveX reference property as the value of an object parameter, TestStand passes the value of the property. Otherwise, TestStand passes a reference for the property object. The DLL function can use the property object reference in conjunction with the TestStand ActiveX API to get and set the values of properties in the object, to add properties to the

object, and so on. Refer to the *TestStand ActiveX API Overview* in the *TestStand ActiveX API* online help for more information on using object references in code modules.

Source Code Tab

Figure 12-4 shows the Source Code tab of the Edit DLL Call dialog box.

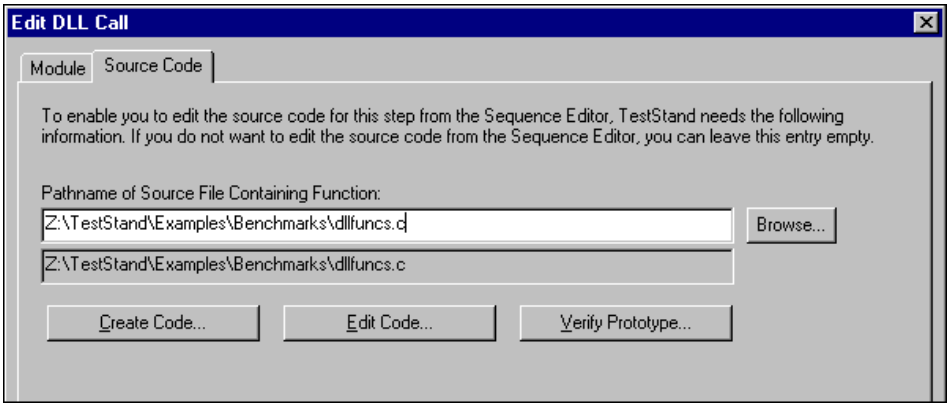

Figure 12-4. Specify Module Dialog Box for DLL Flexible Prototype Adapter—Source Code Tab

You can use the Source Code tab to generate the source code for the DLL function, to edit the source code, or to resolve differences between the parameter list in the source code and the parameter information in the Module tab. You do not have to use the Source code tab for TestStand to call the step code module.

Enter the pathname of the source file in the Pathname of Source File Containing Function control. If you want to create a new source file, you must enter an absolute pathname. If you are using an existing source file, you can enter an absolute or relative pathname. Relative pathnames are relative to the TestStand search directory paths. You can customize the TestStand search directory paths with the **Search Directories** command in the **Configure** menu of the sequence editor menu bar.

To create the source code shell for the function, click on the **Create Code** button. If the file does not already exist, the adapter creates it. If the file already exists, the adapter appends the function to the end of the file. If the function already exists in the file, a dialog box appears that gives you the choice of replacing the current function or adding the new function shell above the current function.

If template source code exists for the step type you are using for the step, the adapter uses the parameter list in the template source code in the new function shell. It also uses the template parameter list to fill in the parameter information on the Module tab. If the Module tab already contains parameter information that disagrees with the parameter list in the template, the adapter displays a dialog box in which you can resolve the conflict.

If the step type has no template or uses the default template, which has an empty parameter list, the adapter uses the parameter information on the Module tab to fill in the parameter list for the new function shell.

After the adapter creates the code, it starts the application that is currently registered on your system for the type of the file, such as `.c`, and displays the file in the application.

If you already have the source code for the function and you want to edit it, click on the **Edit Code** button.

If the parameter list of the function in the source code does not agree with the parameter information on the module tab, the adapter displays a dialog box in which you can resolve the conflict. You can check for any conflicts between the source code and the parameter information on the Module tab by clicking on the **Verify Prototype** button.

When the adapter parses the parameter list in the source code, it has to interpret the parameter declarations. The return value and each parameter must have one of the numeric, array, or object types discussed earlier in this section. Some C parameter declarations can be ambiguous. For instance, `char *` and `char []` can each represent either a null-terminated string or a fixed-size character array.

Table 12-3 indicates how the adapter interprets ambiguous declarations.

Table 12-3. Adapter Interpretation of Ambiguous Declarations

C Data Type in Parameter Declaration in Source Code	Parameter Information in Module Tab
`char *`	Type: C String
`char []`	Type: C String Buffer. Number of Elements: –1
`char [`*nnn*`]`, where *nnn* is a numeric literal	Type: C String Buffer. Number of Elements: *nnn*.
`wchar_t *`	Type: Unicode String
`wchar_t []`	Type: Unicode String Buffer. Number of Elements: –1
`wchar_t [`*nnn*`]`, where *nnn* is a numeric literal	Type: Unicode String Buffer. Number of Elements: *nnn*.
`int *`	Type: Signed 32-bit integer. Pass: Pointer to value.
`int []`	Type: Array. Data Type: Signed 32-bit integer. Number of Elements: –1
`int [`*nnn*`]`, where *nnn* is a numeric literal	Type: Array. Data Type: Signed 32-bit integer. Number of Elements: *nnn*

The adapter handles the other numeric types in the same way it handles the signed 32-bit integers.

Debugging DLLs

You can debug a DLL that you call using the DLL Flexible Prototype Adapter if you create the DLL for debugging in LabWindows/CVI or another ADE. To do so, you must launch the sequence editor or run-time operator interface from LabWindows/CVI or the other ADE. In LabWindows/CVI, you use the **Select External Process** command in the **Run** menu of the Project window to specify the executable for the sequence

editor or run-time operator interface. You then use the **Run** command to start the executable.

If you select the **Step Into** command in TestStand when execution is currently suspended on a step that calls into a LabWindows/CVI DLL you are debugging, LabWindows/CVI breaks at the first statement in the DLL function.

To step out of a LabWindows/CVI DLL function that you are debugging, select the LabWindows/CVI **Finish Function** command. You also can step out of the function by selecting the LabWindows/CVI **Step Into** or **Step Over** command on the last executable statement of the function. After you step out of the DLL function, TestStand suspends execution on the next step in the sequence. If you select the **Continue** command in LabWindows/CVI, TestStand does not suspend execution when the function call returns.

Refer to the LabWindows/CVI product manuals for more information on debugging DLLs in an external process.

Using MFC in a DLL

Microsoft Foundation Class Library (MFC) places several requirements on DLLs that use the DLL version of the MFC run-time library. If you call functions in a DLL that use the DLL version of the MFC run-time library, make sure that the DLL meets these requirements. Also, if the DLL uses resources such as dialog boxes, make sure that each function you call has the `AFX_MANAGE_STATE` macro at the beginning of its function body. Refer to your MFC documentation for more information.

LabVIEW Standard Prototype Adapter

The LabVIEW Standard Prototype Adapter allows you to call any LabVIEW VI that has the specific structure the adapter requires. The LabVIEW Standard Prototype Adapter runs VI code modules using a LabVIEW ActiveX server. The server can be the LabVIEW development environment or a LabVIEW-built application that has the LabVIEW ActiveX server enabled.

LabVIEW Standard Prototype Adapter Module Structure

Code modules for the LabVIEW Standard Prototype Adapter are VIs that contain a specific set of controls and indicators that you assign to a connector pane terminal. The controls and indicators must have names and data types that match the LabVIEW Standard Prototype Adapter parameter

list. TestStand does not require a particular connector pane pattern or that the controls and indicators be assigned to specific terminals. It only requires that you assign them to some terminal in the connector pane of the VI.

You usually create new VIs from the Specify Module dialog box for a step that uses the LabVIEW Standard Prototype Adapter. In this case, TestStand creates the required controls and automatically assigns them to connector pane terminals for you.

Before calling a VI, the adapter assigns values from TestStand to the controls that you wire to the connector pane. After calling the VI, the adapter copies values from the indicators to properties of the TestStand step. The adapter copies each value into its corresponding property when the property exists and the VI does not change the value of the property directly through the TestStand ActiveX API.

A VI must contain a `Test Data` cluster and an `error out` cluster that is wired to the connector pane. A VI also can contain optional controls, which include `Input Buffer`, `Invocation Info`, and `Sequence Context`. The following sections discuss each of the required and optional VI controls.

Test Data Cluster

The LabVIEW Standard Prototype Adapter uses the `Test Data` cluster for returning result data from the VI to TestStand. TestStand can use the data to make a PASS/FAIL determination.

Figure 12-5 shows the Test Data cluster.

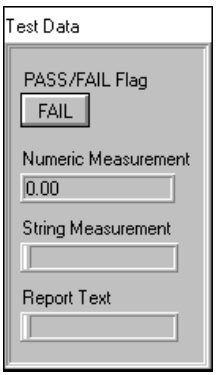

Figure 12-5. Test Data Cluster

Table 12-4 lists the elements of the `Test Data` cluster, their types, and descriptions of how the adapter uses them.

Table 12-4. Test Data Cluster Elements

Name	Type	Description
`PASS/FAIL Flag`	TF	The test VI sets this to indicate whether the test passed. Valid values are `True(PASS)` or `False(FAIL)`. The adapter copies its value into the `Step.Result.PassFail` property if the property exists.
`Numeric Measurement`	DBL	Numeric measurement that the test VI returns. The adapter copies this value into the `Step.Result.Numeric` property if the property exists.
`String Measurement`	abc	String value that the test function returns. The adapter copies the string into the `Step.Result.String` property, if the property exists.
`Report Text`	abc	Output message to display in the report. The adapter copies the message value into the `Step.Result.ReportText` property, if the property exists.

The LabVIEW Standard Prototype Adapter also supports an older version of the `Test Data` cluster from the LabVIEW Test Executive product. The `Test Data` cluster in the LabVIEW Test Executive does not contain a `Report Text` element. Instead, the cluster contains two string elements, `User Output` and `Comment`.

Table 12-5 lists these elements of the older `Test Data` cluster, their types, and description of how the adapter uses them.

Table 12-5. Old Test Data Cluster Elements from LabVIEW Test Executive

Name	Type	Description
`Comment`	abc	Output message to display in the report. The adapter copies the message value into the `Step.Result.ReportText` property if the property exists.
`User Output`	abc	String value that the test function returns. The adapter dynamically creates the step property `Step.Result.UserOutput`, and copies the string value to the step property.

Error Out Cluster

TestStand uses the contents of the `error out` cluster to determine if a run-time error has occurred and takes appropriate action if necessary. When you create a VI, use the standard LabVIEW `error out` cluster as shown in Figure 12-6.

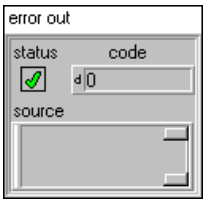

Figure 12-6. Error Out Cluster

Table 12-6 lists the elements of the `error out` cluster, their types, and descriptions of how the adapter uses them.

Table 12-6. Error Out Cluster Elements

Name	Type	Description
`status`	TF	The test VI must set this to `True` if an error occurs. The adapter copies the output value into the `Step.Result.Error.Occurred` property, if the property exists.
`code`	I32	The test VI can set this to a non-zero value if an error occurs.
`source`	abc	The test VI can set this to a descriptive string if an error occurs.

Input Buffer

You can use the `Input buffer` string control to pass input data directly to the VI. The LabVIEW Standard Prototype Adapter automatically copies the contents of the `Step.InBuf` property to the `Input buffer` control if the property exists.

Invocation Information

You can use the `Invocation Information` cluster control to pass additional information to the VI. Figure 12-7 shows the `Invocation Information` control.

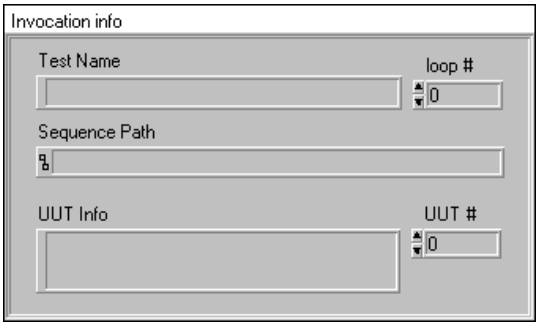

Figure 12-7. Invocation Information Cluster

Table 12-7 lists the elements of the `Invocation Information` cluster, their types, and descriptions of how the adapter assigns a value to each cluster element.

Table 12-7. Error Out Cluster Elements

Name	Type	Description
`Test Name`	abc	Contains the name of the step that invokes the VI.
`loop #`	I32	Contains the loop count if the step that invokes the VI is looping on the step.
`Sequence Path`		Contains the name and absolute path of the sequence file that is running the VI.
`UUT Info`	abc	Contains the value of the `RunState.Root.Locals.UUT.SerialNumber` property, if the property exists.
`UUT #`	I32	Contains the value of the `RunState.Root.Locals.UUT.UUTLoopIndex` property, if the property exists.

Sequence Context

You can use the `Sequence Context` control to obtain a reference to the TestStand sequence context object. You can use the sequence context to access all the objects, variables, and properties in the execution. Figure 12-8 shows the `Sequence Context` control. Refer to the *TestStand ActiveX API Reference* online help for more information on using the sequence context from a VI.

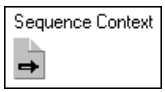

Figure 12-8. Sequence Context Control

Configuring the LabVIEW Standard Prototype Adapter

Figure 12-9 shows the LabVIEW Adapter Configuration dialog box.

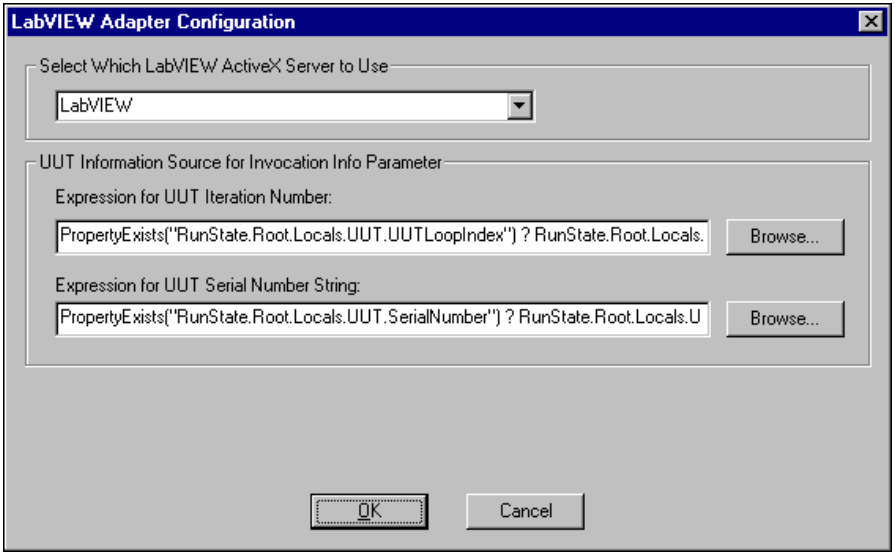

Figure 12-9. LabVIEW Adapter Configuration Dialog Box

The LabVIEW Standard Prototype Adapter runs VIs using a LabVIEW ActiveX server. The server can be the LabVIEW development environment or a LabVIEW-built application that includes the LabVIEW ActiveX server. You specify which server the adapter uses in the Select Which LabVIEW ActiveX Server to Use ring control.

TestStand installs a prebuilt executable with source files for a LabVIEW run-time server in the `TestStand\Components\NI\RuntimeServers\LabVIEW` directory tree. TestStand registers this ActiveX server under the `TestStandLVRTS` ProgID. Refer to the *Customizing and Distributing a LabVIEW Run-Time Server* section in Chapter 16, *Distributing TestStand*, for more information.

The UUT Information Source section contains two controls. The Expression for UUT Iteration Number control specifies the expression that the adapter evaluates at run-time to generate a value to pass to the `UUT #` element of the `Invocation Information` cluster. The Expression for UUT Serial Number String control specifies the expression that the adapter evaluates at run-time to generate a value to pass to the `UUT Info` element of the `Invocation Information` cluster.

Specifying a LabVIEW Standard Prototype Adapter Module

The Specify Module dialog box for the LabVIEW Standard Prototype Adapter is called the Edit LabVIEW VI Call dialog box. Figure 12-10 shows the Edit LabVIEW VI Call dialog box.

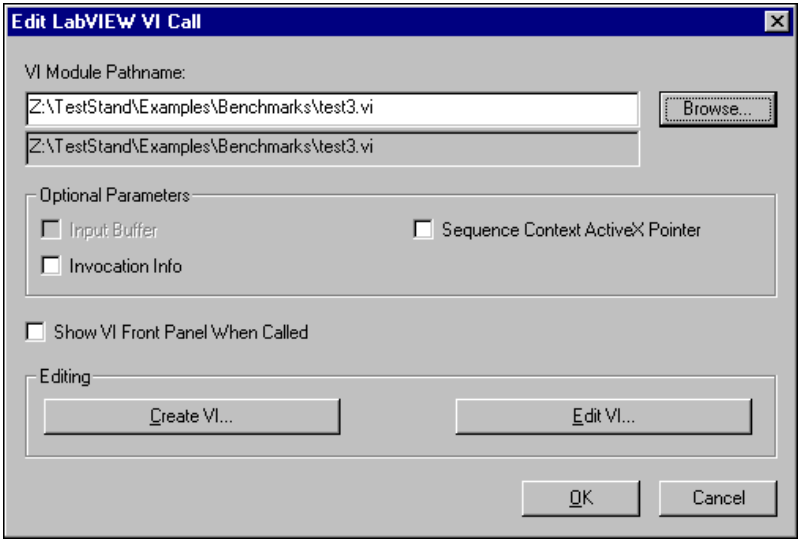

Figure 12-10. Specify Module Dialog Box for LabVIEW Standard Prototype Adapter

The VI Module Pathname control specifies the path and name of the VI that the step calls. If you want the step to call a VI in a LabVIEW `.llb` file, you cannot browse into a `.llb` file. Instead, you must select the `.llb` file and manually append the name of the VI to the pathname, for example, `C:\MyVIs\MyTest.llb\MyTestVI.vi`. You can specify an absolute or relative pathname for the file. Relative pathnames are relative to the TestStand search directory paths. You can customize the TestStand search directory paths with the **Configure»Search Directories** command in the sequence editor menu bar.

The Optional Parameters section contains a checkbox for each of the optional parameters that you can wire to the connector pane of the VI. The checkbox controls are Input Buffer, Invocation Info, and Sequence Context ActiveX Pointer. The Input Buffer control is dim if the `Step.InBuf` property does not exist for the step you are editing. For example, the Input Buffer control is dim for an Action step.

Normally, when the adapter executes a step that calls a LabVIEW VI, the adapter does not activate the front panel of the VI. If you want the adapter

to activate the front panel of the VI, enable the Show VI Front Panel When Called control. The adapter returns the state of the front panel to its original visibility state after the VI finishes executing.

To create a code shell for the VI, click on the **Create Code** button. If the VI file that you specify does not already exist, the adapter creates it. If the file already exists, the adapter prompts you to replace the file. If a VI code template file exists for the step type you are using for the step, the adapter uses the template to create the new VI.

If the VI already exists and you want to edit it, click on the **Edit Code** button.

Debugging a LabVIEW Standard Prototype Adapter Module

To debug a VI while executing the VI from TestStand, you must configure the adapter to use the LabVIEW development environment as the LabVIEW server. There are two ways you can suspend the execution of a VI in LabVIEW.

- Before executing the VI, you can load the VI into LabVIEW and place it in a pause state by clicking on the **Pause** icon button.
- You can select the **Step Into** command in TestStand when execution is currently suspended on a step that calls into a LabVIEW VI.

When LabVIEW suspends a VI, LabVIEW displays the front panel for the VI as shown in Figure 12-11.

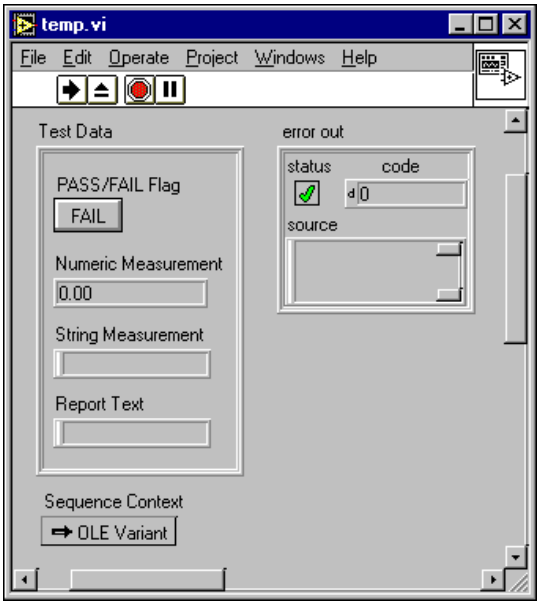

Figure 12-11. Stepping into a LabVIEW VI

A suspended front panel has four main icon buttons:

- Run

- Return to Caller

- Abort Execution

- Pause

From a suspended front panel, you can run the VI multiple times before returning to the calling TestStand step. You can debug the VI by opening the VI diagram and using the standard LabVIEW debug tools. After you are done debugging the VI, you must click on the **Return to Caller** button to return to the calling step and suspend the execution on the next step. If you click on the **Abort Execution** button, the adapter returns a run-time error to the calling step. When you abort the VI execution, the adapter sets the step property `Step.Result.Error.Occurred` to `True`. It also sets the `Step.Result.Error.Code` and `Step.Result.Error.Msg` properties equal to the ActiveX Automation error that the LabVIEW server returns.

C/CVI Standard Prototype Adapter

The C/CVI Standard Prototype Adapter allows you to call any C function that has the TestStand standard C parameter list. The function can be in an object file, library file, or DLL. It also can be in a source file that is in the project that you are currently using in the LabWindows/CVI development environment.

C/CVI Standard Adapter Module Prototypes

The C/CVI Standard Prototype Adapter supports two prototypes, a standard and extended prototype. TestStand provides the extended prototype for backward compatibility with the LabWindows/CVI Test Executive Version 2.0 and earlier. The extended prototype has an additional string parameter. National Instruments recommends that you use the standard prototype unless you have a good reason not to do so.

The standard prototype is

```
void TX_TEST StandardFunc(tTestData *data,
tTestError *error)
```

The extended prototype is

```
int TX_TEST ExtendedFunc(char *params, tTestData *data,
tTestError *error)
```

These prototypes contain two structure parameters, which the adapter uses to pass values into and out of the code module. Table 12-8 lists the fields in the `tTestData` structure.

Table 12-8. tTestData Structure Member Fields

Field Name	Data Type	In/ Out	Description
`result`	`int`	Out	Set by test function to indicates whether the test passed. Valid values are `PASS` or `FAIL`. The adapter copies its value into the `Step.Result.PassFail` property if the property exists.
`measurement`	`double`	Out	Numeric measurement that the test function returns. The adapter copies this value into the `Step.Result.Numeric` property if the property exists.

Table 12-8. tTestData Structure Member Fields (Continued)

Field Name	Data Type	In/ Out	Description
`inBuffer`	`char *`	In	For passing a string parameter to a test function. The adapter copies the `Step.InBuf` property value into this field if the property exists.
`outBuffer`	`char *`	Out	Output message to display in the report. The adapter copies the message value into the `Step.Result.ReportText` property, if the property exists.
`modPath`	`char * const`	In	Directory path of module that contains the test function. The adapter sets this value before executing the code module.
`modFile`	`char * const`	In	Filename of module that contains the test function. The adapter sets this value before executing the code module.
`hook`	`void *`	In	Reserved (no longer used).
`hookSize`	`int`	In	Reserved (no longer used).
`mallocFuncPtr`	`tMallocPtr const`	In	Contains a function pointer to `malloc`, which a code module must use to allocate memory for any buffer that it assigns to the `inBuffer`, `outBuffer`, and `errorMessage` fields.
`freeFuncPtr`	`tFreePtr`	In	Contains a function pointer to `free`, which a code module must use to free any buffers that the `inBuffer`, `outBuffer`, and `errorMessage` fields point to.
`seqContextDisp`	`struct Idispatch *`	In	Dispatch pointer to the sequence context. NULL if you choose not to pass the sequence context.
`seqContextCVI`	`CAObjHandle`	In	LabWindows/CVI ActiveX Automation handle for the sequence context. `0` if you choose not to pass the sequence context.

Table 12-8. tTestData Structure Member Fields (Continued)

Field Name	Data Type	In/ Out	Description
`stringMeasurement`	`char *`	Out	String value that the test function returns. The adapter copies the string into the `Step.Result.String` property, if the property exists.

`replaceStringFuncPtr`	`tReplaceStringPtr const`	In	Contains a function pointer to `ReplaceString`, which a code module can use to reassign a value to the `inBuffer`, `outBuffer`, and `errorMessage` fields. The `ReplaceString` prototype is as follows:
			`int ReplaceString(`
			`char **destString,`
			`char *srcString)`
			The function return value is non-zero if successful.

`structVersion`	`int`	In	Structure version number. A test module can use this to detect new versions of the structure.

Note ***You can use the sequence context to access all the objects, variables, and properties in the execution. Refer to the*** *TestStand ActiveX API Reference* ***online help for more information on using the sequence context from a C/CVI code module.***

Table 12-9 lists the fields in the `tTestError` structure.

Table 12-9. tTestError Structure Member Fields

Field Name	Data Type	In/ Out	Description
`errorFlag`	`Boolean (int)`	Out	The test function must set this to `True` if an error occurs. The adapter copies the output value into the `Step.Result.Error.Occurred` property if the property exists.
`errorLocation`	`tErrLoc (int)`	Out	Reserved (No longer used).
`errorCode`	`int`	Out	The test function can set this to a non-zero value if an error occurs.
`errorMessage`	`char *`	Out	The test function can set this to a descriptive string if an error occurs.

Before calling a code module, the adapter assigns values from TestStand to input fields of the `tTestData` structure.

After calling the code module, the adapter copies the values of the output fields of the structures to properties of the step. The adapter copies a value into a property when the property exists and the code module does not change the value of the property directly through the TestStand ActiveX API. In some cases, the adapter translates the value of a structure field to a different value in the corresponding property.

Table 12-10 lists all the properties that the adapter updates and the value translation, if any, that it makes.

Table 12-10. Step Properties Updated by C/CVI Standard Prototype Adapter

Structure Member	Valid Values that Tests Can Return	Step.Result Property	Step Property Value
`result`	`PASS` or `FAIL`	`PassFail`	`True/False`
`outBuffer`	*`string value`*	`ReportText`	*`string value`*
`measurement`	*`floating-point value`*	`Numeric`	*`numeric value`*
`stringMeasurement`	*`string value`*	`String`	*`string value`*
`errorFlag`	`True` or `False`	`Error.Occurred`	`True/False`
`errorCode`	*`integer value`*	`Error.Code`	*`numeric value`*
`errorMessage`	*`string value`*	`Error.Msg`	*`string value`*

Example C/CVI Standard Prototype Code Module

When you create a code module for the C/CVI Standard Prototype Adapter, you must add the `<TestStand>\Bin\stdtst.h` header file to your source file. The `stdtst.h` file includes the type definitions for the `tTestData` and `tTestError` structures. The following is an example code module that uses the C/CVI standard prototype.

```
// Simple test example
#include "stdtst.h"

void __stdcall __declspec(dllexport) FunctionName (tTestData *testData,
tTestError *testError)
{
   int error = 0;

   // REPLACE THE FOLLOWING WITH YOUR SPECIFIC TEST CODE
   // double measurement = 5.0;
   // char *lastUserName = NULL;

   // testData->measurement = measurement;
```

```
   // The following code shows how to accesses the step properties via
   // the ActiveX Automation API
   // if ((error = TS_PropertyGetValString(testData->seqContextCVI, NULL,
   //   "StationGlobals.TE.LastUserName", 0, lastUserName)) < 0)
   //   goto Error;

Error:
   // FREE RESOURCES
   // if (lastUserName != NULL)
   //   CA_FreeMemory(lastUserName);

   // If an error occurred, set the error flag to cause a run-time error
   // in TestStand.
   if (error < 0)
      {
      testError->errorFlag = TRUE;

      // OPTIONALLY SET THE ERROR CODE AND STRING
      // testError->errorCode = error;
      // testData->replaceStringFuncPtr(&testError->errorMessage,
      //     "A run-time error occurred.");
   }

   return;
}
```

Specifying a C/CVI Standard Prototype Adapter Module

The Specify Module dialog box for the C/CVI Standard Prototype Adapter is called the Edit C/CVI Module Call dialog box. The Edit C/CVI Module Call dialog box contains a Module tab and a Source Code tab. The Module tab specifies the code module that the adapter executes for the step, and the Source Code tab contains additional information that TestStand requires if you want to create and edit a code module in LabWindows/CVI.

Figure 12-12 shows the Module tab on the Edit C/CVI Module Call dialog box.

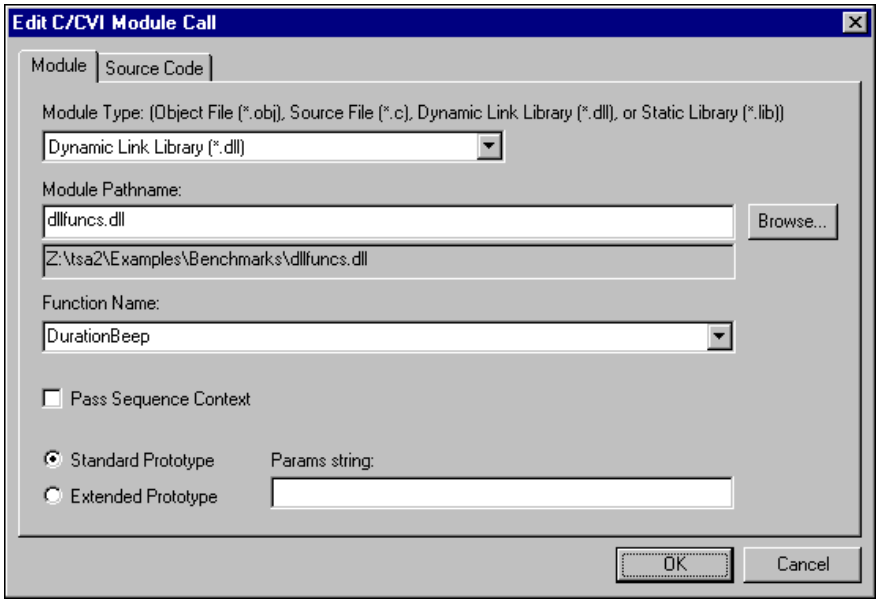

Figure 12-12. Specify Module Dialog Box for C/CVI Standard Prototype Adapter—Module Tab

- **Module Type**—Selects the type of code module the step calls. The adapter supports calling functions in C source files, object files, dynamic link library files, and static library files.
- **Module Pathname**—Specifies the pathname of the code module file that contains the function the step calls. You can specify an absolute or relative pathname for the module file. Relative pathnames are relative to the TestStand search directory paths. You can customize the TestStand search directory paths using the **Search Directories** command in the **Configure** menu of the sequence editor menu bar.
- **Function Name**—Selects the function in the code module that the step calls. If the code module is a DLL file, and the DLL file contains a type library, the adapter automatically populates the function ring with all the function names in the type library. Otherwise, the adapter attempts to read the code module file and finds the names of all functions.
- **Standard Prototype** and **Extended Prototype**—Select the function prototype for the function. Use the Params String control to specify the value of the extra parameter for the extended prototype.

- **Pass Sequence Context**—Specifies whether the adapter passes a sequence context to the code module. The adapter passes the sequence context in two forms in the `tTestData` structure. It passes it as an CVI ActiveX Automation handle in the `seqContextCVI` field and as an ActiveX Automation dispatch pointer in the `seqContextDisp` field. Enable the checkbox if you want to call the TestStand ActiveX API in the code module or if the code module must pass the sequence context as a dispatch pointer to another function.

Figure 12-13 shows the Source Code tab for the Edit C/CVI Module Call dialog box.

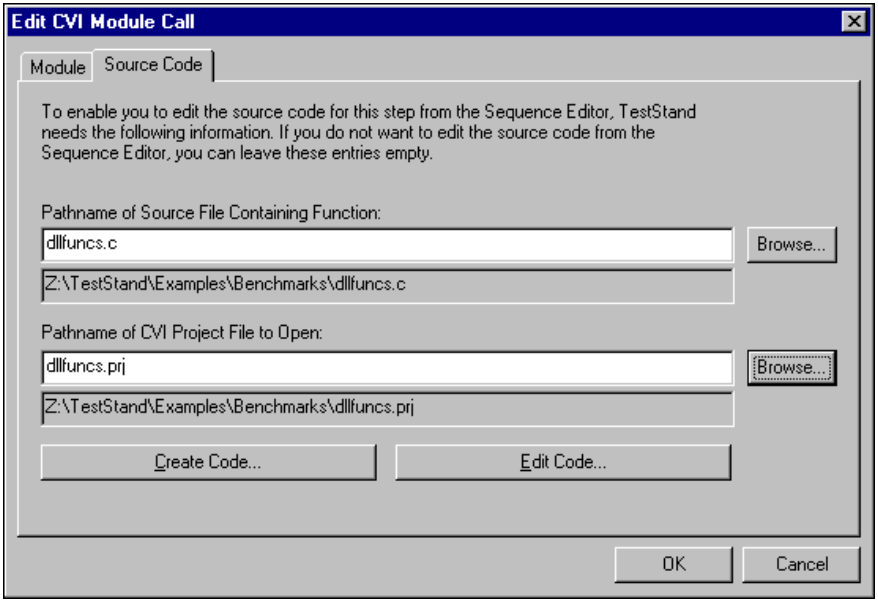

Figure 12-13. Specify Module Dialog Box for C/CVI Standard Prototype Adapter—Source Code Tab

You can use the Source Code tab to generate or edit the source code for the function. You do not have to use the Source code tab for TestStand to call the step code module.

Enter the pathname of the source file in the Pathname of Source File Containing Function control. If you want to create a new source file, you must enter an absolute pathname. If you are using an existing source file, you can enter an absolute or relative pathname. Relative pathnames are relative to the TestStand search directories. You can customize the

TestStand search directory paths with the **Search Directories** command in the **Configure** menu of the sequence editor menu bar.

If the code module is a DLL or static library, you must enter the name of the LabWindows/CVI project that you used to create the DLL or static library file. If the code module is an object module, you can specify a project if you want to.

To create the source code shell for the function, click on the **Create Code** button. If the source file you specify does not already exist, the adapter creates it. If the file already exists, the adapter appends the function to the end of the file. If a source code template file exists for the step type you are using for the step, the adapter uses the template to create the shell of the new function. If the project file you specify does not already exist, the adapter creates it and adds the source file to it.

If you already have the source code for the function, and you want to edit it, click on the **Edit Code** button.

When you use the **Create Code** or **Edit Code** button, the adapter starts a copy of LabWindows/CVI and opens the source file. If you specify a project file in the Source Code tab, the adapter also opens the project in LabWindows/CVI. When you use the **Create Code** button and the function already exists in the file, a dialog box appears giving you the choice of replacing the current function or adding the new function shell above the current function.

Note ***You cannot use the* Create Code *button when you select the Extended Prototype.***

The adapter can launch two different copies of LabWindows/CVI. The adapter uses one copy to execute test modules and to let you edit source files. The adapter always opens the `TestStand\Bin\tecvirun.prj` project in this copy of LabWindows/CVI. The adapter uses the other copy of LabWindows/CVI to let you edit the projects you use to create DLLs, static libraries, and object files.

Configuring the C/CVI Standard Prototype Adapter

You can specify whether the C/CVI Standard Prototype Adapter executes tests *in-process* or *out-of-process*. When the adapter runs tests in-process, it executes them in the same process as the sequence editor or operator interface you are running. When the adapter runs tests out-of-process, it executes them in an external instance of the LabWindows/CVI development environment. You specify this option by using the **Adapters** command in the **Configure** menu.

Figure 12-14 shows the configuration dialog box for the C/CVI Standard Prototype Adapter.

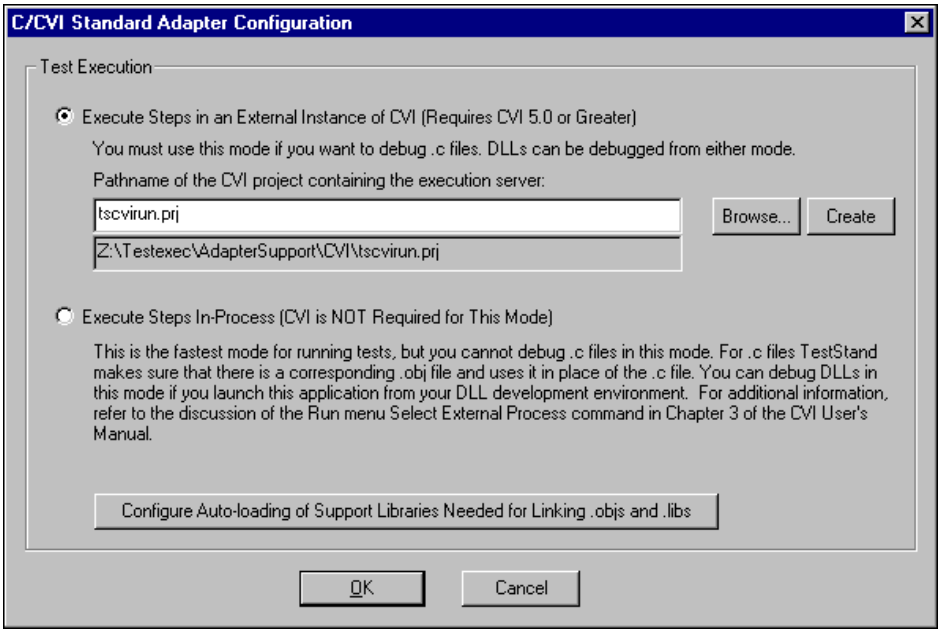

Figure 12-14. C/CVI Standard Adapter Configuration Dialog Box

Executing Code Modules In-Process

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

Object and Library Code Modules

When the adapter loads an object or static library file, the LabWindows/CVI Run-time Engine resolves all external references in the file. When running tests in-process, the adapter must load the support libraries that the object file or static library file depends on before it loads the file. You can configure a list of support libraries for the adapter to load by copying them manually to the `TestStand\AdapterSupport\CVI\AutoLoadLibs` directory, or by clicking on the **Configure Auto-loading of Support Libraries Needed for Linking .objs and .libs** button on the C/CVI Standard Adapter Configuration dialog box. When you click on the **Configure Auto-Loading** button, the Auto-Load Library Configuration dialog box appears as shown in Figure 12-15. Before you can configure the

auto-load libraries, you must unload all code modules. The adapter prompts you to do this.

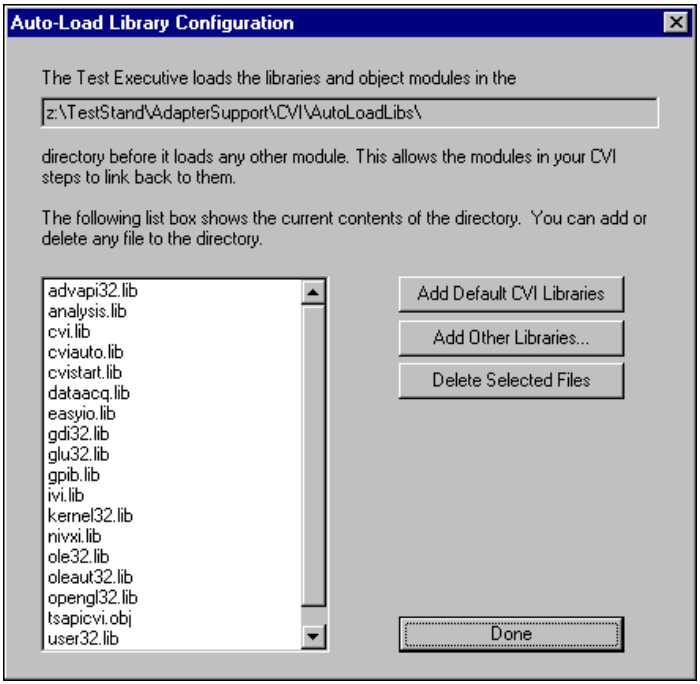

Figure 12-15. Auto-Load Library Configuration Dialog Box

The **Add Default CVI Libraries** button searches for an installation of the LabWindows/CVI development environment and copies the LabWindows/CVI static library files to the auto-load library directory.

You can select the **Browse** button to search for files to copy to the auto-load library directory.

The **Delete Selected Files** button removes the selected files from the auto-load library directory.

Source Code Modules

When executing tests in-process, the adapter cannot directly execute code modules that are in C source files. Instead, the adapter attempts to find an object file by the same name. If the adapter finds the object file, the adapter executes the code in the object file. If the adapter cannot find the object file, the adapter prompts you to create the object file in an external version of

LabWindows/CVI. If you decline to create the object module, the adapter reports a run-time error.

Debugging a DLL Code Module

When executing code modules in-process, the only code modules you can debug are code modules in DLLs that you create for debugging in LabWindows/CVI or another ADE. To do so, you must launch the sequence editor or run-time operator interface from LabWindows/CVI or the other ADE. In LabWindows/CVI, you use the **Select External Process** command in the **Run** menu of the Project window to specify the executable for the sequence editor or run-time operator interface. You then use the **Run** command to start the executable.

If you select the **Step Into** command in TestStand when execution is currently suspended on a step that calls into a LabWindows DLL you are debugging, LabWindows/CVI breaks at the first statement in the DLL function.

To step out of a LabWindows/CVI DLL function that you are debugging, select the LabWindows/CVI **Finish Function** command. Also, you can step out of the function by selecting the LabWindows/CVI **Step Into** or **Step Over** command on the last executable statement of the function. After you step out of the DLL function, TestStand suspends execution on the next step in the sequence. If you select the **Continue** command in LabWindows/CVI, TestStand does not suspend execution when the function call returns.

Refer to the LabWindows/CVI product manuals for more information on debugging DLLs in an external process.

Executing Code Modules in an External Instance of LabWindows/CVI

To execute tests in an external instance of LabWindows/CVI, the adapter launches a copy of LabWindows/CVI and loads an execution server project in LabWindows/CVI. You can specify the execution server project to load in the C/CVI Standard Adapter Configuration dialog box. The default project is `TestStand\bin\tecvirun.prj`.

When a TestStand step calls a function in an object, static library, or DLL file, the execution server project automatically loads the file and executes the function in the external instance of LabWindows/CVI.

If you want a TestStand step to call a function in a C source file, you must include the C source file in the execution server project before you run the project. Also, you must include all support libraries other than LabWindows/CVI libraries in the project.

Debugging C Source and DLL Code Modules

When the adapter executes tests in an external instance of LabWindows/CVI, you can debug C source and DLL code modules. To debug DLL code modules, you must create the DLL in LabWindows/CVI with the DLL Debugging option enabled. LabWindows/CVI honors all breakpoints you set in the source files for the DLL project.

If you select the **Step Into** command in TestStand when execution is currently suspended on a step that calls into the DLL, LabWindows/CVI breaks at the first statement in the DLL function.

To step out of a LabWindows/CVI DLL function that you are debugging, select the LabWindows/CVI **Finish Function** command. You also can step out of the function by selecting the LabWindows/CVI **Step Into** or **Step Over** command on the last executable statement of the function. After you step out of the DLL function, TestStand suspends execution on the next step in the sequence. If you select the **Continue** command in LabWindows/CVI, TestStand does not suspend execution when the function call returns.

Sequence Adapter

The Sequence Adapter allows you to call subsequences with parameters. 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 use the default value that the subsequence defines for the parameter.

Usually, you use the Sequence Call built-in step type to call sequences, but you can use the Sequence Adapter from any step type that can use module adapters, such as Pass/Fail Test or Numeric Limit Test. Using a Sequence Call step is the same as using an Action step with the Sequence Adapter.

After the sequence call step executes, the Sequence Adapter can set the step status. If the sequence that 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 the same properties in subsequence step that generated the run-time error. If the sequence call is successful, the adapter does not set the step status. Depending on the type of step, the resulting status is `Done` or `Passed`.

You can define the parameters for a sequence on the Parameters tab of the Sequence File window. Figure 12-16 shows the Parameters tab.

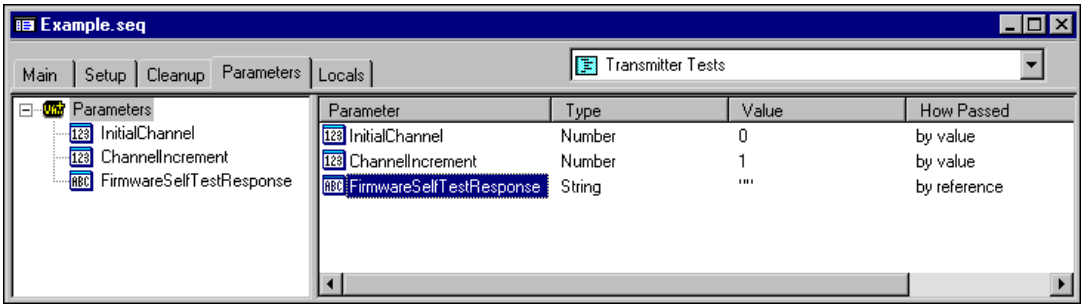

Figure 12-16. Example Sequence Parameters

The Parameters tab defines each parameter name, its TestStand data type, its default value, and whether you pass the argument by value or by reference. For more information on sequence file parameters, refer to the *Parameters Tab* section in Chapter 5, *Sequence Files*.

Specifying a Sequence Adapter Module

The Specify Module dialog box for the Sequence Adapter is called the Edit Sequence Call dialog box. This section describes the contents of the Edit Sequence Call tab and the Remote Execution tab.

Edit Sequence Call Tab

Figure 12-17 shows the Edit Sequence Call dialog box.

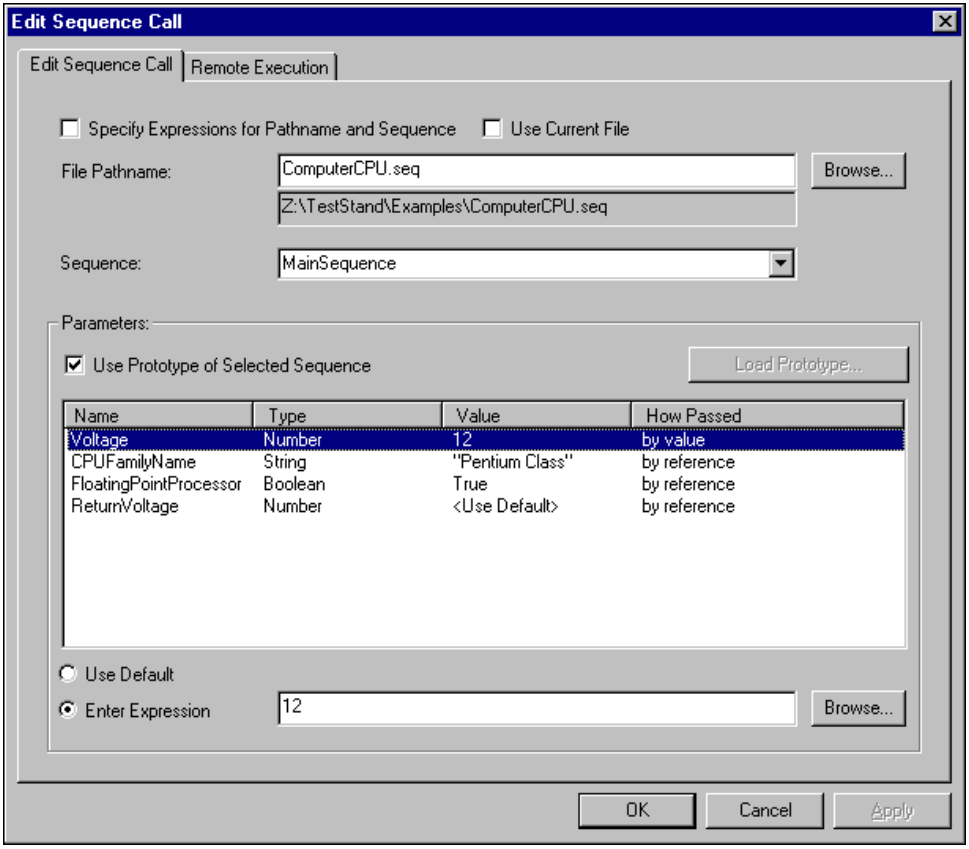

Figure 12-17. Specify Module Dialog Box for the Sequence Adapter—Edit Sequence Call Tab

The Specify Module Dialog Box for the Sequence Adapter contains the following controls:

- **Specify Expressions for Pathname and Sequence**—Selects whether you specify the sequence name and the sequence file pathname through literal strings or through expressions that TestStand evaluates at run-time. When you use literal strings, you enter the actual pathname of the sequence file in the File Pathname control.
- **Sequence**—Contains the names of the sequences in the sequence file you specify. When you use expressions, the File Path Expression and Sequence Expression controls appear in place of the File Pathname and

Sequence controls. You use these controls to specify the expressions for the sequence file pathname and the sequence name.

- **Use Current File**—Enable this checkbox if you want to call a sequence in the sequence file that you are currently editing. The File Pathname or File Path Expression control dims when you enable the Use Current File checkbox.
- **List Box in the Parameters Section**—Displays the parameters that the step passes to the sequence. For each parameter, the list box shows the name of the parameter, its TestStand data type, the value the step passes to the sequence, and whether the step passes the argument by value or by reference.

  The contents of the list box must be consistent with the parameter definitions in the sequence that the step calls. You must extract the parameter definitions from the sequence or from another sequence that has the same parameter list.

If you disable the Specify Expressions for Pathname and Sequence checkbox, you can extract the parameter list directly from the sequence that you select in the Sequence ring. To do so, enable the Use Prototype of Selected Sequence option. If you enable Use Prototype of Selected Sequence, the contents of the parameter list box updates whenever you select a different sequence from the Sequence ring.

To specify the value that the step passes for a parameter, select a parameter and enter the value in the Enter Expression control. You can specify an expression that TestStand evaluates at run-time. If you want to use the default value that the sequence defines for the parameter, enable the Use Default control. The parameter definition in the sequence determines whether the step passes the argument by value or by reference.

If you enable the Specify Expressions for Pathname and Sequence option, you must use the **Load Prototype** button to load a prototype from a sequence that has the same parameter list definition as the sequences that the step might call.

For a parameter that the step passes by value, the expression in the Enter Expression control must evaluate to a value that is compatible with the data type of the parameter. For a parameter that the step passes by reference, you also can enter an expression that evaluates to a value. If so, the adapter passes a reference to a copy of the value, and it discards the value that the sequence returns in the copy. If you want to receive the value that the sequence returns, you must specify the name of a station global variable, sequence file global variable, sequence parameter, or step property.

Although the parameter list that the step uses must be consistent with the parameter list that the sequence defines, the step can specify fewer parameters than the sequence specifies. The data types for the parameters in the step must be compatible with the corresponding parameters in the sequence. The adapter uses the default values for the parameters that the step does not pass explicitly.

Note ***When calling a sequence on a remote host, you can pass single-valued properties or arrays of number, Boolean, and string properties. You can pass these properties by value or by reference. You also can pass container properties or ActiveX reference properties to a remote sequence if the receiving parameter type is an ActiveX reference property.***

Remote Execution Tab

You can configure a step that uses the Sequence Adapter to invoke a sequence in a TestStand engine that runs on a remote host as a server. Figure 12-18 shows the Remote Execution tab for the Edit Sequence Call dialog box.

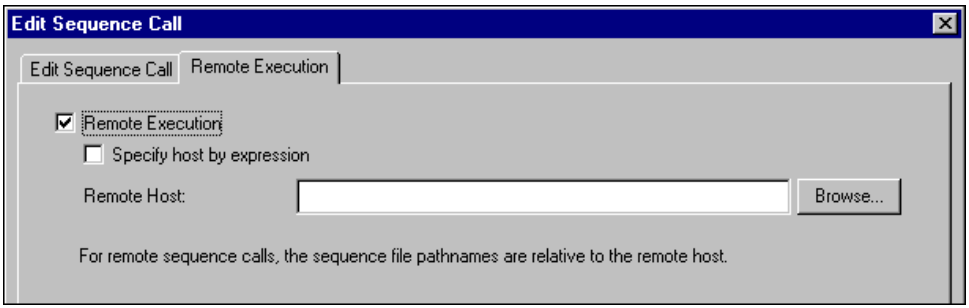

Figure 12-18. Specify Module Dialog Box for the Sequence Adapter—Remote Execution Tab

The Remote Execution tab contains the following options:

- **Remote Execution**—Selects whether TestStand executes the sequence call on a remote host.
- **Specify host by expression**—Selects whether you specify the remote host name through literal strings or through expressions that TestStand evaluates at run-time. When disabled, you can use the **Browse** button to select a remote host name on the network. When enabled, you can use the **Browse** button to build an expression.
- **Remote Host**—Contains the name of remote host.

When you specify a sequence file pathname on the Edit Sequence Call tab and you enable the step for remote execution, TestStand locates the sequence file according to the type of path, as shown in Table 12-11.

Table 12-11. Path Resolution of Sequence Pathnames for Remotely Executed Steps

Type of Path	Where Found When Editing	Where Found When Running	Example
Relative	In the TestStand search paths you configure on the client (local) machine.	In the TestStand search paths you configure on the server (remote) machine.	`Transmit.seq`
Absolute	On the client (local) machine.	On the server (remote) machine.	`C:\Projects\Transmit.seq`
Network	On the machine specified in the network path	On the machine specified in the network path	`\\Remote\Transmit.seq`

When you edit a step in a sequence file on a client 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 system. When you run the step on the client, TestStand resolves the path for the sequence file on the server system.

You have three ways of managing your remote sequence files for remote execution.

1. You can add a common pathname to the search paths for the client and the server so that each resolves to the same relative pathname.
2. You can duplicate the files on your client and server system so that the client edits an identical file to the file that the server runs.

3. You can use absolute paths that specify a mapped network drive or full network path so that the file the client edits and the file the server runs is the same sequence file.

When you execute a remote sequence, you cannot single-step or set breakpoints in the remote sequence. If tracing is enabled, 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 includes only the sequences that run on the remote system. If you want to access properties from the client sequence context, you must pass the property objects or their values as parameters to the remote sequence. You can use the ActiveX API to access properties within a property object.

Setting up TestStand as a Server for Remote Execution

If you want TestStand to invoke a sequence on a remote TestStand server host, you must properly configure the server on the remote system. You must enable the TestStand server to accept remote execution requests, the server must be registered with the operating system, and you must configure the Windows system security to allow users to access and launch the server remotely.

You can enable the remote server to accept remote execution requests from a client machine by enabling the Enable Remote Execution option on the Remote Execution tab of the Station Options dialog box.

A TestStand server is active when the TestStand application `TestStand\bin\rengine.exe` is running on a remote system. Each TestStand client communicates with a dedicated version of the remote engine application. In Windows NT, the remote server launches automatically each time a TestStand client uses the server. In Windows 95 and 98, you must launch the remote server manually, and only one client is able to use the server at a time. You can automatically launch the server by placing a shortcut to the application in the startup folder on the server system.

TestStand automatically registers the server during installation. If you want to manually register or unregister the server, you can invoke the executable with the `/RegServer` and `/UnregServer` command-line arguments respectively.

Before a client can communicate with a server, you must configure the security permissions for the server on the Windows system of the server.

For Windows NT, you must complete the following steps to configure the security permissions for the server.

1. Login using a userid that has administrator privileges.
2. Run `dcomcnfg` from the command line, which displays the Distributed COM Configuration Properties application window.
3. On the Default Properties tab, make sure that the Enable Distributed COM on this computer option is checked.
4. On the Applications tab, select TestStand Remote Engine and then click the **Properties** button. On the Identity tab of the TestStand Remote Engine Properties dialog box, make sure that the Interactive User option is selected.
5. You must give permission to the appropriate users so that they can access the remote server. You should give everyone access permissions and appropriate users launch permission. Only users who have launch permission will be able to access the server. You can do this by one of two ways.
   - You can specify the default security on the Default Security tab of the Distributed COM Configuration Properties application window.
   - You can give individual users access to the server. On the Applications tab, select TestStand Remote Engine and then click the **Properties** button. Use the Security tab of the TestStand Remote Engine Properties dialog box to configure the permissions for a specific server.

For Windows 95 or Windows 98, you must complete the following steps to configure the security permissions for the server.

1. You must configure your Windows network options in the system control panel to use User-level access control to use distributed COM.
2. Run `dcomcnfg` from the command line, which displays the Distributed COM Configuration Properties application window.
3. On the Default Properties tab, make sure that the Enable Distributed COM on this Computer option is checked.
4. On the Default Security tab, make sure that the Enable Remote Connection option is checked.
5. You must give permission to the appropriate users so that they can access the remote server. You can do this by one of two ways:
   - You can specify the default security on the Default Security tab of the Distributed COM Configuration Properties application window.

– You can give individual users access to the server. You can do this by selecting the server name, TestStand Remote Engine, on the Applications tab and then clicking on the **Properties** button. On the Security tab of the TestStand Remote Engine Properties dialog box, you can add users to a list for access to the server.

ActiveX Automation Adapter

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

Configuring the ActiveX Automation Adapter

When you specify the module for an Automation step, TestStand stores the IDs and names of the object and member that the step calls. During execution, the Automation Adapter must invoke the ActiveX Automation server and specify which object to create and which member to call. You can configure the adapter to use either the IDs (early binding) or names (late binding) to specify to the server what operations to perform on what object. The Automation Adapter Configuration dialog box contains a single checkbox option, Use Late Binding. When you enable this option, the adapter uses late binding during execution. Otherwise it uses early binding.

Early binding is more efficient, but requires that the IDs for objects and methods exposed by automation servers do not change. If you are developing an automation server in an ADE that does not provide direct control over IDs, it is recommended that you use late binding during development so that inadvertent changes to IDs do not unnecessarily invalidate the module information for the step. When you finish developing your automation server, uncheck this option and update the IDs in the client sequences. You can update the IDs in the client sequences by either editing each step's module information or by running the **Tools»Update Automation Identifiers** command on each sequence file containing Automation steps that reference your server.

If you are using a third-party or release version of an automation server, or you are developing a server in an ADE that allows you to control your server’s IDs, it is recommended that you uncheck the Use Late Binding option.

When you configure the adapter to use late binding, the Automation Adapter uses the stored names to determine the proper IDs to use at run-time. The Automation Adapter looks in the most recent version of the server's type information. Servers also can specify type information in different languages (locales). If the Automation Adapter can not find a version of the type information that uses the system default language ID, it attempts to find type information that uses the English or Neutral language IDs, in that order.

Specifying an ActiveX Automation Adapter Module

The Specify Module dialog box for the ActiveX Automation Adapter is called the Edit Automation Call dialog box. Figure 12-19 shows the Edit Automation Call dialog box.

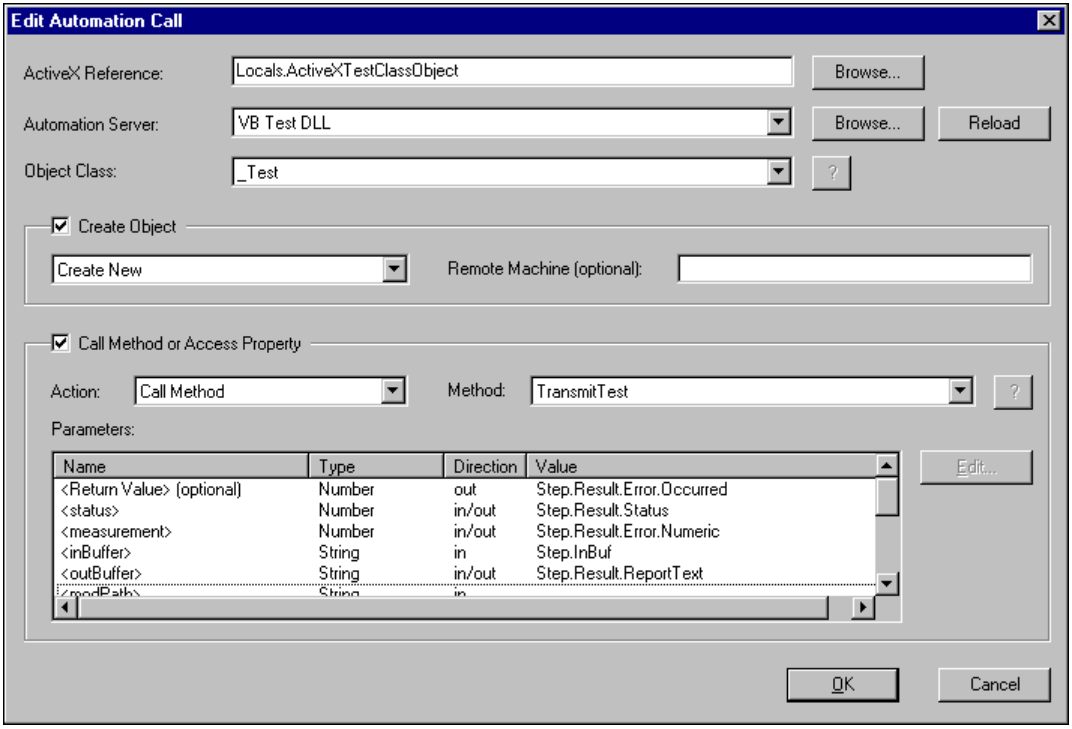

Figure 12-19. Specify Module Dialog Box for ActiveX Automation Adapter

The Edit Automation Call dialog box contains the following controls.

- **ActiveX Reference**—This control specifies a variable or property of type ActiveX Reference. When a step creates an object, the adapter assigns the object reference to the variable or property, if specified. Otherwise, the adapter automatically releases the object reference after executing the step. If the step does not create an object, but instead calls a method or accesses a property, the ActiveX Reference control must contain the value of a valid ActiveX reference, usually from a previously executed ActiveX Automation Adapter step. You can use the **Browse** button to display the Expression Browser dialog box.
- **Automation Server**—This ring control specifies the name of the server that the step uses. The adapter populates the ring control with a list of the ActiveX automation servers registered with Windows. You can select a server from the list by clicking on the ring control, or you can use the **Browse** button to load a type library file from disk for a specific server. TestStand registers the type library with Windows. If you want the adapter to refresh the list of registered servers and their type library information, you can click on the **Reload** button. You also can refresh server type library information when you select **File»Unload All Modules**.
- **Object Class**—This ring control specifies the name of the server class that the step uses when creating, or invoking an object of that class. When you select a server, the adapter populates the Object Class control with a list of objects defined for that server. The ring control separates the list of objects into two groups separated by a line. The upper group includes all top-level objects that the adapter can create. The lower group includes all other objects that must be created by the server as a result of an invocation of a method or get property call. If a server type library contains help strings or links to a help file for a class, you can click on the **?** button to display the help.
- **Create Object**—You can enable this section of the dialog box to specify whether the step creates a new instance of the object class when the adapter executes the step. When the step creates an object and you specify a property name in the ActiveX Reference control, the adapter assigns the value of the object handle to the property. Otherwise, the adapter automatically releases the object reference after executing the step.

When you create an object you can select one of the following options:

– **Create New**—Use this option to create a new object and obtain a reference to the object. If the server application is already running, this option may or may not start another copy of the application. This is determined by the server application.

– **Attach to Active**—Use this option to get a reference to an active Application object.

– **Create From File**—Use this option to load an existing object from a file, and obtain a reference to the object. If the server application is already running, this option may or may not start another copy of the application. This is determined by the server application. When you make this selection, the dialog box displays a file selection control and a **Browse** button. Use these controls to specify the path of the file.

Use the Remote Machine control to optionally specify the remote system to create the object on. The control is dim when you select Attach to Active.

- **Call Method or Access Property**—You can enable this section of the dialog box to specify the class method that the step invokes or the class property that the step accesses. The Type control lists the types of access that the server defines for the selected object class. The options include Call Method, Set Property, and Get Property. For example, if an object class does not have any methods, the control does not list the Call Method option.

  Once you select the type of access, the adapter populates the Member control with the method or property names that the class defines for the access type. If a server type library contains help strings or links to a help file for a method or property, you can select the **?** button to display the help.

- **Parameters**—This control contains the input and output parameters for the selected method or property. If the selected access type is Get Property, the control usually contains a single output parameter. If the access type is Set Property, the control usually contains a single input parameter. When invoking a method, the control contains any number of input and output parameters. The Value field is automatically populated for parameters that have default values.

  You can specify an expression for each parameter by double-clicking on the parameter or by clicking on the parameter and selecting the **Edit** button. When you make this selection, TestStand displays the Edit Parameter Value dialog box.

Figure 12-20 shows the Edit Parameter Value dialog box.

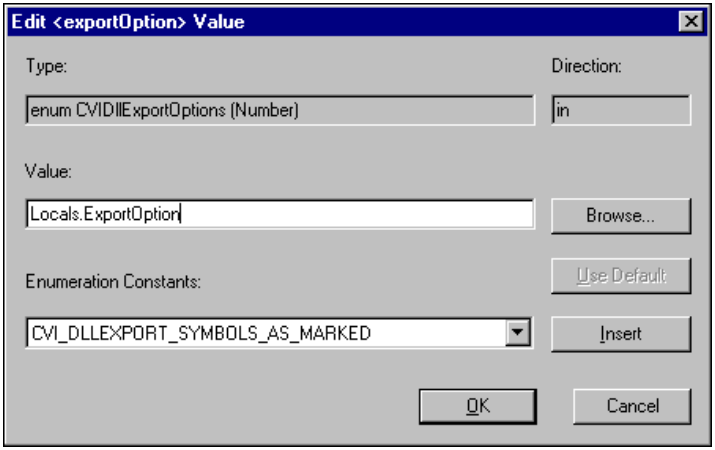

Figure 12-20. Edit Parameter Value Dialog Box

In the Edit Parameter Value dialog box, the Type and Direction controls indicate the ActiveX data type for the parameter and whether the parameter is input, output, or both. In the Value control you specify the parameter argument. For input parameters, you must specify a value for its argument. If an input parameter has a default value, you can select the **Use Default** button to instruct the adapter to use the default value specified by the server. For optional parameters, you can leave the Value control empty or specify the name of a variable, parameter, or property. If the type library defines enumeration constants for input parameters, you can select a constant from the Enumeration Constants control and click on the **Insert** button to copy the constant name to the expression in the Value control. The server indicates which input parameters are optional. TestStand marks all method output parameters as optional.

The ActiveX Automation Adapter supports the Variant data types shown in Table 12-12.

Table 12-12. Variant Data Types Supported by the ActiveX Automation Adapter

Variant Type	Variant Description
`VT_EMPTY`	nothing
`VT_NULL`	SQL style Null
`VT_I2`	2 byte signed int
`VT_I4`	4 byte signed int
`VT_R4`	4 byte real
`VT_R8`	8 byte real
`VT_CY`	currency
`VT_DATE`	date
`VT_BSTR`	OLE Automation string
`VT_DISPATCH`	`Idispatch FAR*`
`VT_ERROR`	`SCODE`
`VT_BOOL`	True=`-1`, False=`0`
`VT_VARIANT`	`VARIANT FAR*`
`VT_UNKNOWN`	`IUnknown FAR*`
`VT_UI1`	`unsigned char`
`VT_ARRAY`	`SAFEARRAY*`

Note ***TestStand does not support variant structures which are available with DCOM for Windows 95, Windows 98, and Windows NT 4.0 SP4. You can assign a value only from a variant of type*** `VT_DISPATCH` ***and*** `VT_UNKNOWN` ***to a TestStand ActiveX Reference data type. TestStand uses the*** `VariantChangeType` ***OLE function to convert data types between OLE variants and TestStand variables and properties.***

The TestStand ActiveX Automation Adapter does not support handling events generated by an ActiveX automation server.

Running and Debugging ActiveX Automation Servers

To debug an out-of-process executable server, you must launch the automation server in the ADE that you created it in and independently launch the sequence editor or run-time operator interface. If you want to debug an in-process DLL server, you usually launch the sequence editor or run-time operator interface from the ADE. When using Visual Basic, you can use the Run with Full Compile option to debug a DLL in-process server. Refer to your ADE documentation for more information on debugging ActiveX Automation servers.

Using ActiveX Servers with TestStand

This section discusses using ActiveX servers with TestStand.

Registering a Server

You can register an ActiveX Automation server DLL by calling the Windows executable `regsvr32.exe` with the DLL pathname as the command-line argument. You can unregister the DLL server by calling `regsvr32.exe` with `/u` and the DLL pathname as the command-line argument.

You usually can register an ActiveX Automation server executable by running the server executable with the `/RegServer` command-line argument. To unregister an executable server, call the executable with the `/UnregServer` command-line argument.

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

Compatibility Issues with Visual Basic

If you are developing an automation server in an ADE that does not provide direct control over IDs, you must ensure that the adapter can find the server identifiers or names defined in a TestStand step. When you rebuild an ActiveX Automation server in Visual Basic, you can select one of three compatibility options. Depending on the level of compatibility, and the changes made to a project, Visual Basic compiles an appropriate new

server, which can contain new identifiers. Visual Basic has the following compatibility options.

- **No compatibility**—When you rebuild a server with this option, the new server maintains no compatibility with a previously compiled server. Visual Basic generates new unique identifiers for the server, which prevents any previously compiled client application that uses early binding from working properly with the server. When you rebuild a server with this option set, Visual Basic changes the ID used to uniquely identify the server's type information. TestStand therefore cannot properly update an Automation Adapter step regardless of whether the adapter is configured for early or late binding. This setting is not recommended for use with TestStand.
- **Project compatibility**—You usually use this option when working with multiple projects under development within Visual Basic. It is not meant to assure compatibility with non-Visual Basic compiled client applications that use early binding. You can use the project compatibility option only after you build the server DLL or executable once. When you rebuild a server with this option set, Visual Basic does not change the ID used to uniquely identify the server's type information. TestStand therefore is able to use the type information to determine the IDs associated with the names stored in the step. It is recommended that you configure the Automation Adapter to use late binding when creating a server using this option.
- **Binary compatibility**—When you use this option Visual Basic attempts to maintain compatibility with compiled client application that use early binding. If you remove a member from the server, Visual Basic can no longer maintain binary compatibility. You can use the binary compatibility option only after you build the server DLL or executable once. When you rebuild a server with this option, Visual Basic maintains the IDs used to identify objects and methods. TestStand therefore is able to use the IDs stored in the step without accessing the type information at run time. It is recommended that you configure the Automation Adapter to use early binding when creating a server that uses this option.

National Instruments recommends that when you are developing and debugging a Visual Basic ActiveX Automation server in conjunction with developing sequences within TestStand, you should use the Project Compatibility option in Visual Basic and configure the ActiveX Automation Adapter to use late binding. This ensures that the ActiveX Automation adapter can properly find and invoke the server after you recompile the server. Once you define the interface for the server, you should recompile the project using only binary compatibility. You can then

use the **Tools»Update Automation Identifiers** command to respecify steps to use the final server identifiers. Once you properly update the automation identifiers in your steps, you can enable the ActiveX Automation Adapter to use early binding.

For more information on creating and debugging Visual Basic ActiveX automation servers, refer to your Visual Basic documentation and the article, “Building, Versioning, and Maintaining Visual Basic Components,” by Ivo Salmre, *Microsoft Developer Network*, Microsoft Corporation, February 1998.

13

Process Models

This chapter discusses the default process model that TestStand includes. It also describes the directory structure that TestStand uses for process model files and the special capabilities that the TestStand sequence editor has for editing process model sequence files.

You can best understand the contents of this chapter if you have already read the *Process Models* section in Chapter 1, *TestStand Architecture Overview*, which discusses the purpose of process models, model callbacks, and entry points, and the relationship between a process model and a client sequence file. This chapter does not repeat that information.

Directory Structure for Process Model Files

The TestStand installer places the files for the default process model files under the following directory.

`TestStand\Components\NI\Models\TestStandModel`

The default process model consists of a process model sequence file and several supporting sequence files. The name of the process model sequence file is `TestStandModel.seq`

If you want to modify the default process model, copy the process model sequence file and its supporting files to a different subdirectory, and rename the process model sequence file. It is best to copy the files to a subdirectory under the `TestStand\Components\User\Models` directory. The subdirectories of `TestStand\Components\NI` contain various TestStand files that you might want to modify or replace. If you modify these files directly, the installers for newer versions of TestStand might overwrite your customizations. Consequently, it is best to keep the files that you create or modify separate from the files that TestStand installs. For this purpose, TestStand includes the subdirectories under `TestStand\Components\User` in its list of search paths. The TestStand installer creates a subdirectory tree under `TestStand\Components\User` for you. Not only do you use the subdirectories to protect your customized components, you also use them as the staging area for the components that you include in your own run-time distribution of TestStand.

National Instruments recommends that you place each process model under its own subdirectory under `TestStand\Components\User\Models`.

If you customize the default process model and want to name it MyProcess, copy the default process model files to `TestStand\Components\User\Models\MyProcess`, and rename the process model sequence file `MyProcess.seq`. You must also establish your custom process model as the process model for the station using the Model tab of the Station Options dialog box.

Special Editing Capabilities for Process Model Sequence Files

The TestStand sequence editor has specific features for creating or modifying process model sequence files.

If you want TestStand to treat a sequence file as a process model, you must mark it as a process model file. To do so, select **Sequence File Properties** from the **Edit** menu. In the Sequence File Properties dialog box, select the Advanced tab. In the Advanced tab, select the **Model** entry in the Type ring control.

Figure 13-1 shows the settings for a process model file in the Advanced tab of the Sequence File Properties dialog box.

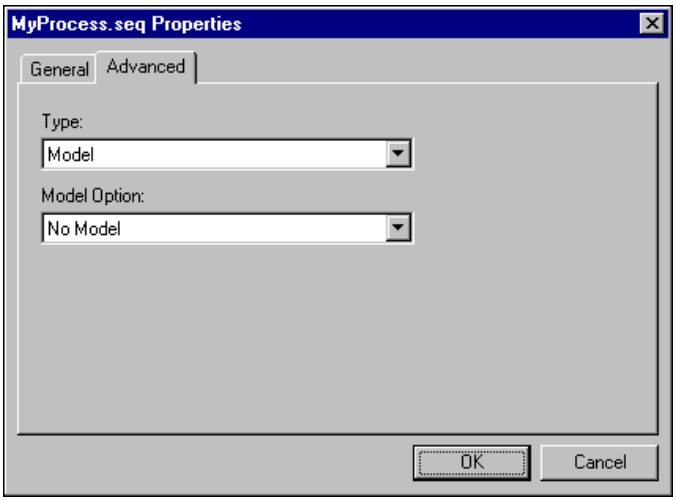

Figure 13-1. Process Model Settings in the Advanced Tab of the Sequence File Dialog Box

Although you edit a process model sequence file in a regular Sequence File window, the file has special contents. In particular, some of the sequences in the files are model entry points, and some are model callbacks. TestStand maintains special properties for the entry point and callback sequences. You can specify the values of these properties when you edit the sequences in a process model file. When you display the Sequence Properties dialog box for any sequence in a model file, the dialog box contains a Model tab.

Sequence Properties Model Tab

You access the Sequence Properties dialog box by selecting the **Sequence Properties** item from the context menu in a step list of an individual sequence view or by selecting the **Properties** item from the context menu for a sequence in the All Sequences view. If the sequence file is a process model file, the dialog box contains a Model tab. The first control on the Model tab is the Type ring control.

Figure 13-2 shows the pull-down menu for the Type ring control.

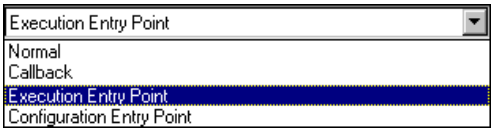

Figure 13-2. Type Ring Control in the Sequence Properties Model Tab

The Type ring control lists the different types of sequences that a process model file can contain. The following sections describe the different types of sequences.

Normal Sequences

A *normal* sequence is any sequence *other* than a callback or entry point. In a process model file, you use normal sequences as utility subsequences that the entry points or callbacks call.

When you select the **Normal** entry in the Types ring, nothing else appears on the Model tab

Callback Sequences

Model callbacks are sequences that entry point sequences call and that the client sequence file can override. By marking sequences in a process model file as callbacks, you specify the set of process model operations a sequence developer can customize. When editing the client file, the sequence developer can override the callback by selecting **Edit»Sequence File Callbacks**. Refer to the *Sequence View Context Menu* section in Chapter 5, *Sequence Files*, for more information on using the Sequence File Callbacks dialog box.

Some model callbacks have full implementations. For example, the `TestReport` callback in the default process model is sufficient to handle most types of test results. Other model callbacks are merely placeholders that you override with sequences in the client file. For example, the `MainSequence` callback in the model file is a placeholder for the `MainSequence` callback in the client file.

When you select the Callback entry in the Type ring, the Copy Steps and Locals when Creating an Overriding Sequence checkbox appears. This checkbox determines what TestStand does when you click on the **Add** button in the Sequence File Callbacks dialog box to create an overriding sequence in the client file. If you enable the checkbox, TestStand copies all the steps and local variables in the callback sequence in the model file to the callback sequence you create in the client file. TestStand always copies the sequence parameters regardless of the checkbox setting.

Entry Point Sequences

Entry point sequences are sequences you can invoke from the menus in the TestStand sequence editor or from an operator interface program. You can specify two different types of entry points:

- *Execution entry points*—Use this entry point to run test programs. Execution entry points call the `MainSequence` callback in the client file. The default process model contains two execution entry points: `Test UUTs` and `Single Pass`. By default, execution entry points appear in the **Execute** menu. Execution entry points appear in the menu only when the active window contains a sequence file that has a `MainSequence` callback.
- *Configuration entry points*—Use this entry point to configure a feature of the process model. Configuration entry points usually save the configuration information in a `.ini` file in the `TestStand\cfg` directory. By default, configuration entry points appear in the **Configure** menu. For example, the default process model contains the

configuration entry point, `Config Report Options`. The `Config Report Options` entry point appears as **Report Options** in the **Configure** menu.

When you select Execution Entry Point or Configuration Entry Point from the Type ring, numerous controls appear on the Model tab. The contents of the Model tab are the same for all types of entry points.

Figure 13-3 shows the contents of the Model tab for the `Test UUTs` execution entry point.

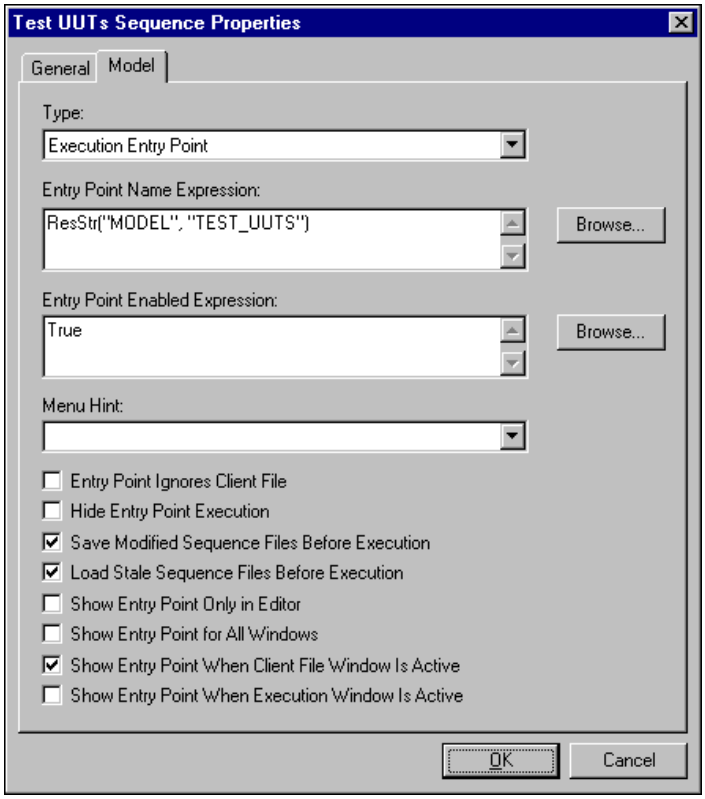

Figure 13-3. Model Tab for an Execution Entry Point Sequence

The Model tab for an Execution Entry Point Sequence contains the following controls:

- **Entry Point Name Expression**—Use this control to specify a string expression for the menu item name of the entry point. If you specify a literal string for the menu item name, you must enclose it in double quotes. If you want to store the name in a string resource file, you can use the `GetResourceString` expression function to retrieve the name from the file. Refer to the *Expressions* section in Chapter 8, *Sequence Context and Expressions*, for more information.
- **Entry Point Enabled Expression**—Use this control to specify a Boolean expression that TestStand evaluates to determine whether to enable the menu item for the entry point. If the expression evaluates to `False`, TestStand dims the entry point in the menu. If the expression is empty, the entry point is enabled in the menu.
- **Menu Hint**—Use this control to specify a menu for the entry point. If you leave the Menu Hint control empty, TestStand uses the default menu for the entry point type. Click on the arrow at the right edge of the control to pull down a menu that contains the following entries: File, Edit, View, Execute, Debug, Configure, Window, and Help.

  You can enter one or more names directly in the control. If you specify multiple names, you must separate them with commas. TestStand uses the first menu name in the list that it can find in the operator interface. This is useful if you use multiple operator interfaces that have different menu names. If TestStand cannot find any menus in the operator interface with the names that you list in the control, it uses the default menu for the entry point type.
- **Entry Point Ignores Client File**—Enable this option if the sequence does not call the client file and you want the sequence to run without preloading the client file. This option prevents TestStand from preloading the client sequence file when you run the entry point even if the client sequence file is set to preload when execution begins.

  When you run the entry point, TestStand uses the callback implementations in the model file regardless of whether the client file overrides them. The `Config Report Options` entry point uses this option so that you can select **Configure»Report Options** even when TestStand is unable to preload the modules in the active sequence file.

- **Hide Entry Point Execution**—Enable this option if you do not want TestStand to display an Execution window for the execution of the entry point. If you enable this option, you do not see a window for the execution unless a run-time error or breakpoint occurs.
- **Save Modified Sequence Files Before Execution**—Enable this option if you want TestStand to save the contents of windows to disk when you invoke the entry point. If this option is enabled when you run the entry point, TestStand checks all windows that have pathnames. If one or more windows have changes that you have not yet saved, TestStand prompts you to save your changes. If you click on **Yes**, TestStand saves the files.
- **Load Stale Sequence Files Before Execution**—Enable this option if you want TestStand to check the disk dates of files that are in memory when you invoke the entry point. If the current disk date of a file differs from the disk date of when you last loaded or saved it, TestStand gives you the option to reload the file.
- **Show Entry Point Only in Editor**—Enable this option if you want the entry point to appear only in the TestStand sequence editor and not in the run-time operator interfaces.
- **Show Entry Point for All Windows**—Enable this option if you want the entry point to appear in the menu regardless of the type of window, if any, that is currently active. For example, the `Configure Report Options` entry point configures the report options for the model and has no client-specific effects. Thus, you might want to access it from any window or even if no window is active. If you enable this option, TestStand dims the remaining two checkboxes.
- **Show Entry Point When Client File Window is Active**—Enable this option if you want the entry point to appear in the menu when a Sequence File window is the active window. For example, the execution entry points are in the **Execute** menu only when a sequence file is active.
- **Show Entry Point When Execution Window is Active**—Enable this option if you want the entry point to appear in the menu when an Execution window is the active window.

Contents of the Default Process Model

Figure 13-4 shows a list of all the sequences in the default TestStand process model. The first three sequences are entry points. The last sequence is a utility subsequence that the execution entry points call. The other sequences are model callbacks that you can override in a client sequence file.

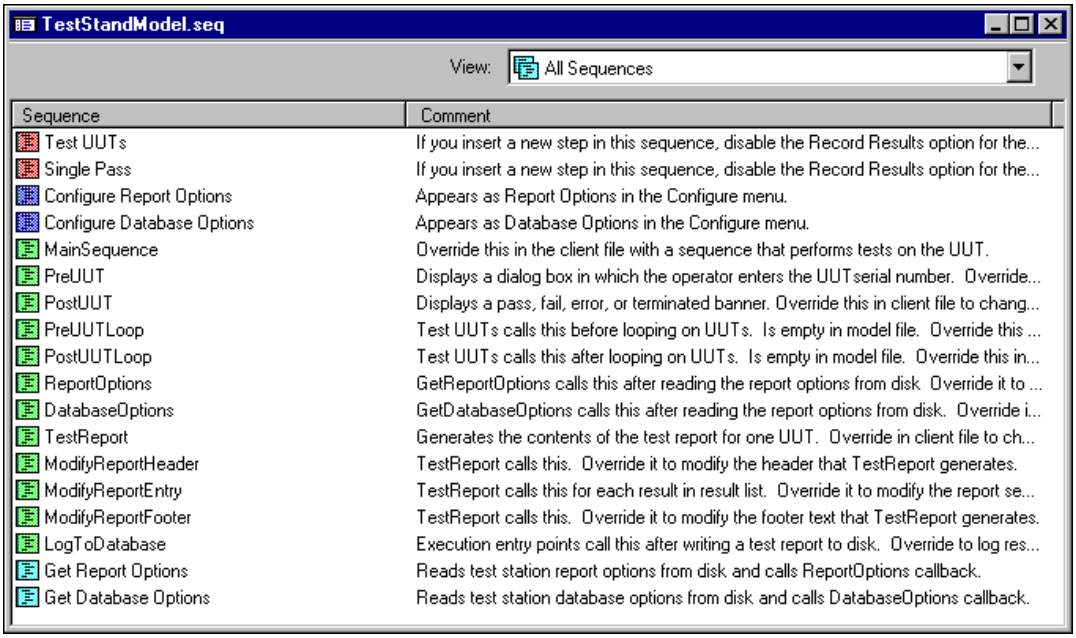

Figure 13-4. List of All Sequences in the Default TestStand Process Model File

The default TestStand process model file contains the following sequences:

- `Test UUTs`—This sequence is an execution entry point that initiates a loop that repeatedly identifies and tests UUTs. When a window for a client sequence file is active, the **Test UUTs** item appears in the **Execute** menu. Refer to the *Test UUTs Entry Point* section later in this chapter for more information.
- `Single Pass`—This sequence is an execution entry point that tests a single UUT without identifying it. In essence, the `Single Pass` entry point performs a single iteration of the loop that the `Test UUTs` entry point performs. When a window for a client sequence file is active, the **Single Pass** item appears in the **Execute** menu.

- `Config Report Options`—This sequence is a configuration entry point that displays a dialog box in which you can specify the contents, format, and pathname of the test report. The settings you make in the dialog box apply to the test station as a whole. The entry point saves the station report options to disk. The entry point appears as **Report Options** in the **Configure** menu. Refer to Chapter 14, *Managing Reports*, for more information on the report options.
- `Configure Database Options`—This sequence is a configuration entry point that displays a dialog box in which you can specify the database logging options. This entry point is a placeholder for database functionality.
- `MainSequence`—This sequence is a model callback that the Test UUTs entry point calls for each UUT. The `MainSequence` callback is empty in the process model file. The client file must contain a `MainSequence` callback that performs the tests on a UUT.
- `PreUUT`—This sequence is a model callback that displays a dialog box in which the operator enters the UUT serial number. The `Test UUTs` entry point calls the `PreUUT` callback at the beginning of each iteration of the UUT loop. If the operator indicates through the dialog box that no more UUTs are available for testing, the UUT loop terminates.
- `PostUUT`—This sequence is a model callback that displays a banner indicating the result of the test that the `MainSequence` callback in the client file performs on the UUT. The `Test UUTs` entry point calls the `PostUUT` callback at the end of each iteration of the UUT loop.
- `PreUUTLoop`—This sequence is a model callback that the `Test UUTs` entry point calls before the UUT loop begins. The `PreUUTLoop` callback in the default process model file is empty.
- `PostUUTLoop`—This sequence is a model callback that the `Test UUTs` entry point calls after the UUT loop terminates. The `PostUUTLoop` callback in the default process model file is empty.
- `ReportOptions`—This sequence is a model callback that the execution entry points call through the `GetReportOptions` subsequence. After reading the test station report options from disk, `GetReportOptions` calls the `ReportOptions` callback to give the client sequence file a chance to modify the report options. For example, you might want to force the report format to be ASCII text for a particular client sequence file. The `ReportOptions` callback in the default process model file is empty.
- `DatabaseOptions`—This sequence is a model callback that the execution entry points call through `GetDatabaseOptions` subsequence. After reading the test station database options from disk,

GetDatabaseOptions calls the DatabaseOptions callback to give the client sequence file a chance to modify the database options. The DatabaseOptions callback in the default process model file is empty. The DatabaseOptions callback is a placeholder for future database functionality.

- TestReport—This sequence is a model callback that the execution entry points call to generate the contents of the test report for one UUT. You can override the TestReport callback in the client file if you want to change its behavior entirely. The default process model defines a test report for a single UUT as consisting of a header, an entry for each step result, and a footer. If you do not override the TestReport callback, you can override the ModifyReportHeader, ModifyReportEntry, and ModifyReportFooter model callbacks to customize the test report.

  The TestReport callback makes the determination as to whether the report body is built with sequences or a DLL based on a setting in the Configure Report Options dialog box. If you select the sequence report generation option, TestReport calls the AddReportBody sequence in either ReportGen_txt.seq or ReportGen_html.seq to build the report body. The sequence report generator uses a series of sequences with steps that recursively process the result list for the execution. If you select the DLL report generation option, TestReport calls a single function in modelsupport.dll to build the entire report body before returning. The project and source code for the LabWindows/CVI built DLL is available. If you select the DLL option, TestStand generates reports faster, but TestStand does not call ModifyReportEntry callbacks.

- ModifyReportHeader—This sequence is a model callback that the TestReport model callback calls so that the client sequence file can modify the report header. ModifyReportHeader receives the following parameters: the UUT, the tentative report header text, and the report options. The ModifyReportHeader callback in the default process model file is empty.

- ModifyReportEntry—This sequence is a model callback that the TestReport model callback calls so that the client sequence file can modify the entry for each step result. Through subsequences, TestReport calls ModifyReportEntry for each result in the result list for the UUT. ModifyReportEntry receives the following parameters: an entry from the result list, the UUT, the tentative report entry text, the report options, and a level number that indicates the call stack depth at the time the step executed. The ModifyReportEntry callback in the default process model file is empty.

Note ***The Report Options dialog box allows you to select between producing the report body using sequences or a DLL. If you select the DLL option, TestStand generates reports faster, but TestStand does not call*** `ModifyReportEntry` ***callbacks.***

- `ModifyReportFooter`—This sequence is a model callback that the `TestReport` model callback calls so that the client sequence file can modify the report footer. `ModifyReportFooter` receives the following parameters: the UUT, the tentative report footer text, and the report options. The `ModifyReportFooter` callback in the default process model file is empty.
- `LogToDatabase`—This sequence is a model callback that the execution entry points call after they write the test report for a UUT to disk. You can use this callback to log the result information for a UUT to a database. `LogToDatabase` receives the following parameters: the UUT, the result list for the UUT, the report options, and the report text. The `LogToDatabase` callback in the default process model file is empty.
- `Get Report Options`—This sequence is a utility sequence that the execution entry points call at the beginning of execution. `GetReportOptions` reads the report options and then calls the `ReportOptions` callback to give you a chance to modify the report options in the client file.
- `Get Database Options`—This sequence is a utility sequence that the execution entry points call at the beginning of execution. `GetDatabaseOptions` reads the database options and then calls the `DatabaseOptions` callback to give you a chance to modify the database options in the client file.

Test UUTs Entry Point

Table 13-1 lists the more significant steps in the `Test UUTs` execution entry point.

Table 13-1. Order of Actions in the Test UUTs Entry Point

Action Number	Description	Remarks
1	Call `PreUUTLoop` model callback.	Callback in model file is empty.
2	Call `GetReportOptions` utility sequence.	Reads station options from disk. Calls `ReportOptions` model callback to allow client to modify options.
3	Call `GetDatabaseOptions` utility sequence.	Reads station options from disk. Calls `DatabaseOptions` model callback to allow client to modify options.
4	Increment the UUT index.	—
5	Call `PreUUT` model callback.	Gets the UUT serial number from the operator.
6	If no more UUTs, go to action 14.	—
7	Determine the report file pathname.	—
8	Clear results list.	—
9	Call `MainSequence` model callback	`MainSequence` callback in client performs the tests on the UUT.
10	Call `PostUUT` model callback.	Displays a pass, fail, error, or terminate banner.
11	Call `TestReport` model callback.	Generates test report for the UUT.
12	Write the UUT report to disk.	Can append to an existing file or create a new file.
13	Call `LogToDatabase` model callback.	Log test results to database for the UUT.
14	Loop back to action 3	—

Table 13-1. Order of Actions in the Test UUTs Entry Point (Continued)

Action Number	Description	Remarks
15	Call `PostUUTLoop` model callback.	Callback in model file is empty.
16	Read test report into memory.	Ensures that the entire test report is in memory so that the sequence editor or operator interface can display it. If you use separate files for each UUT, this action reads only the test report for the last UUT.

Single Pass Entry Point

Table 13-2 lists the more significant steps in the `Single Pass` execution entry point.

Table 13-2. Order of Actions in the Single Pass Entry Point

Action Number	Description	Remarks
1	Call `GetReportOptions` utility sequence.	Reads station options from disk. Calls `ReportOptions` model callback to allow client to modify options.
2	Call `GetDatabaseOptions` utility sequence.	Reads station options from disk. Calls `DatabaseOptions` model callback to allow client to modify options.
3	Determine the report file pathname.	—
4	Clear results list for UUT.	—
5	Call `MainSequence` model callback	`MainSequence` callback in client performs the tests on the UUT.
6	Call `TestReport` model callback.	Generates test report for the UUT.
7	Write the UUT report to disk.	Can append to an existing file or create a new file.
8	Call `LogToDatabase` model callback.	Log test results to database for the UUT.

Support Files for the Default Process Model

Many sequences in the default process model file call functions in DLLs and subsequences in other sequence files. TestStand installs these supporting files and the DLL source files in the same directory that it installs the process model sequence file.

Table 13-3 lists the files that TestStand installs for the default process model in the `TestStand\NI\Models\TestStandModels` directory.

Table 13-3. Default Process Model Files

File Name	Description
`TestStandModel.seq`	Entry point and model callback sequences for the default process model.
`reportgen_html.seq`	Subsequences that add the header, result entries, and footer for a UUT into an HTML test report.
`reportgen_txt.seq`	Subsequences that add the header, result entries, and footer for a UUT into an ASCII text test report.
`modelsupport.dll`	DLL containing C functions that the process model sequences call. Includes functions that display the Report Options dialog box, read and write the report options from disk, determine the report file pathname, obtain the UUT serial number from the operator, and display status banners.
`modelsupport.prj`	LabWindows/CVI project that builds `modelsupport.dll`.
`modelsupport.fp`	LabWindows/CVI function panels for the functions in `modelsupport.dll`.
`modelsupport.h`	C header file that contains declarations for the functions in `modelsupport.dll`.
`modelsupport.lib`	Import library in Visual C/C++ format for `modelsupport.dll`.
`modelpanels.uir`	LabWindows/CVI user interface resource file containing panels that the functions in `modelsupport.dll` use.
`modelpanels.h`	C header file containing declarations for the panels in `modelpanels.uir`.
`main.c`	C source for utility functions.
`banners.c`	C source for functions that display status banners.

Table 13-3. Default Process Model Files (Continued)

File Name	Description
`report.c`	C source for functions that display the Report Options dialog box, read and write the report options from disk, and determine the report file pathname.
`uutdlg.c`	C source for function that obtains the UUT serial number from the operator.
`c_report.c`	C source for generating HTML and ASCII reports for the DLL option on the Report Options dialog box.

You can view the contents of the `reportgen_html.seq` and `reportgen_txt.seq` sequence files in the sequence editor. Notice that each is a *model sequence file* and contains an empty `ModifyReportEntry` callback. Each file has a `PutOneResultInReport` sequence that calls `ModifyReportEntry`. The client sequence file can override the `ModifyReportEntry` callback. TestStand requires that all sequence files that contain direct calls to model callbacks must also contain a definition of the callback sequence and must be model files.

`TestStandModel.seq` also contains an empty `ModifyReportEntry` callback, even though no sequences in `TestStandModel.seq` call `ModifyReportEntry` directly. `TestStandModel.seq` contains a `ModifyReportEntry` callback so that `ModifyReportEntry` appears in the Sequence File Callbacks dialog box for the client sequence file.

14

Managing Reports

This chapter describes how you manage and use test reports in TestStand.

Implementation of the Test Report Capability

Most of the test report capabilities that this chapter describes are not native to the TestStand engine or sequence editor. Instead, the default process model that comes with TestStand implements them. This allows you to customize all aspects of test reports. Refer to the *Contents of the Default Process Model* section in Chapter 13, *Process Models*, for more information.

If you do not modify or replace the test report implementation in the process model, you can still customize the contents of test reports using the Report Options dialog box that the default process model provides. Refer to the *Report Options Dialog Box* section later in this chapter for more information.

The default process model relies on the automatic result collection capability of the TestStand engine to accumulate the raw data for the test report for each UUT. The TestStand engine can automatically collect the result of each step into a result list for an entire sequence. The result list for a sequence contains the result of each step it runs and the result list of each subsequence call it makes. The default process model calls the main sequence in the client sequence file to test a UUT. Thus, the result list that the TestStand engine accumulates for the main sequence contains the raw data for the test report for the UUT. Refer to the *Result Collection* section in Chapter 6, *Sequence Execution*, for information on automatic result collection.

Using Test Reports

The `Test UUTs` and `Single Pass` entry points in the default process model generate UUT test reports. The `Test UUTs` entry point generates a test report and writes it to disk after each pass through the UUT loop. In the Report Options dialog box, you can choose whether to create a separate file for each UUT test report or to aggregate the test reports for all the UUTs that you test during one execution into one file. You can choose whether to generate unique pathnames or reuse the same report file pathnames from one execution to another. You can specify the directory in which to write the test report files, and you can specify whether to include the date and time in the filenames. You can display the Report Options dialog box by selecting **Configure»Report Options**.

In the TestStand sequence editor, the Report tab of the Execution window displays the report for the current execution. Usually, the Report tab is empty until execution completes. The default process model can generate reports in either HTML or ASCII text formats.

The Report tab can display reports in HTML, ASCII text, or Rich Text Format (RTF). You also can use an external application to view reports in these or other formats by selecting the **View»Launch Report Viewer** command when an Execution window is active. You can use the **External Viewers** menu item in the **Configure** menu to specify the external application that TestStand launches to display a particular report format.

Figure 14-1 shows a test report in HTML text format on the Report tab of an Execution window.

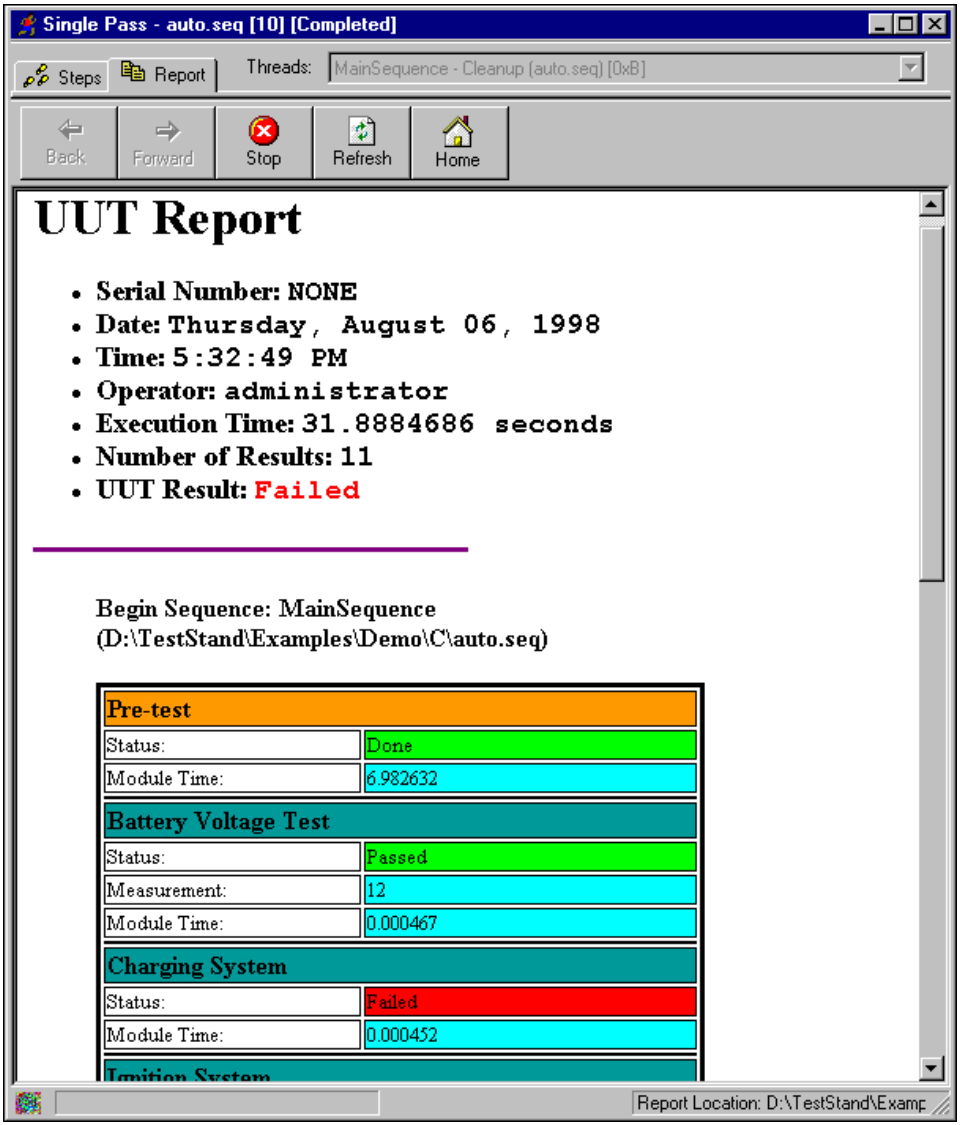

Figure 14-1. HTML Test Report in the Report Tab

Figure 14-2 shows a test report in ASCII format on the Report tab of an Execution window.

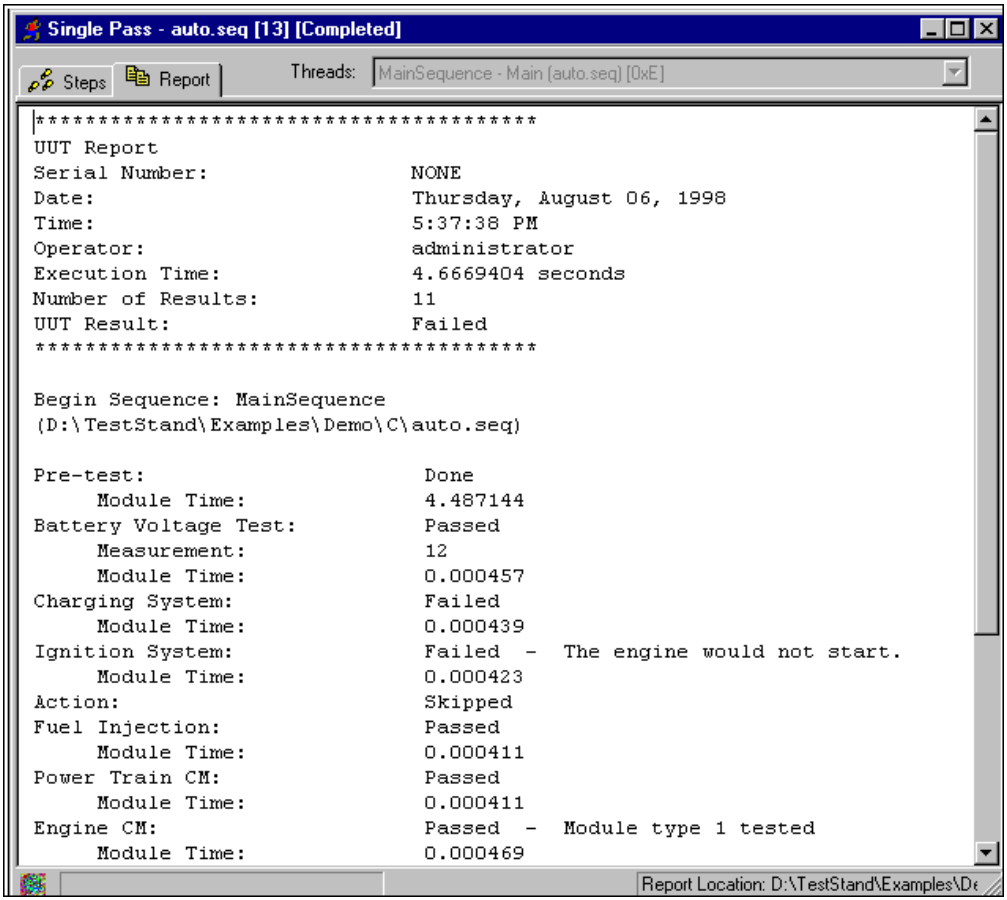

Figure 14-2. ASCII Text Test Report in the Report Tab

Report Options Dialog Box

You can access the Report Options dialog box by selecting **Configure»Report Options**.

In the Report Options dialog box, you can customize the generation of report files. The settings you choose in the Report Options dialog box apply to all sequences that you run on the station using the **Test UUTs** and **Single Pass** items in the **Execute** menu.

When you select the **Report Options** command, TestStand calls the `Config Report Options` entry point in the default process model. Thus, while the dialog box is active in the sequence editor, the Running tag appears on left side of the status bar.

The Report Options dialog box contains two tabs: the Contents tab and the Report File Pathname tab.

Contents Tab

Figure 14-3 shows the Contents tab of the Report Options dialog box.

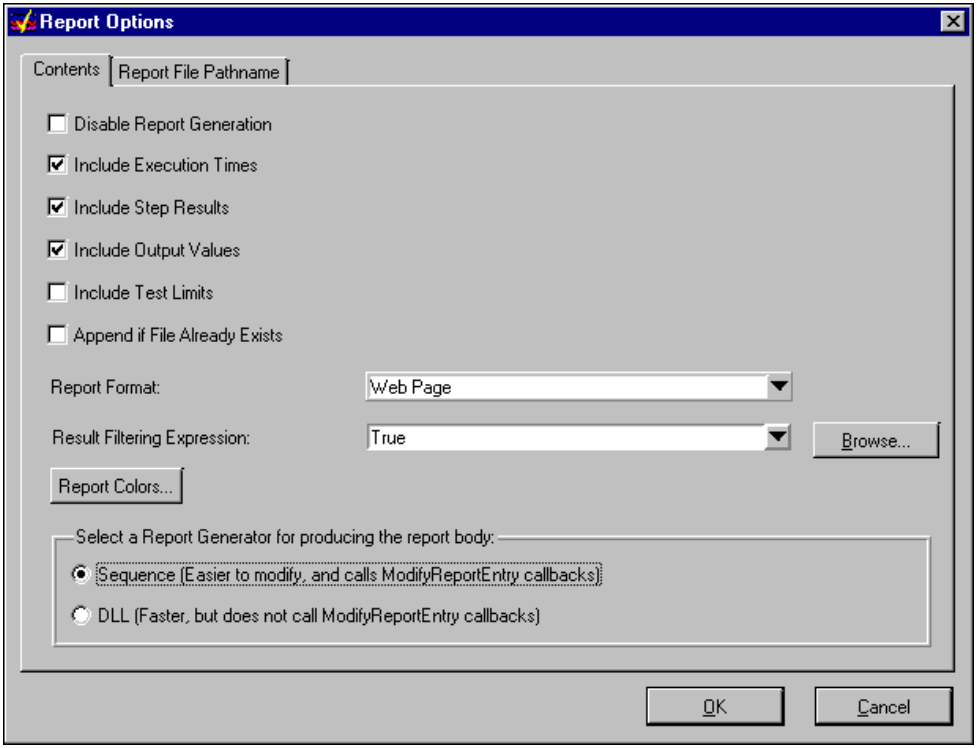

Figure 14-3. Report Options Dialog Box—Contents Tab

The Contents tab of the Report Options dialog box contains the following options:

- **Disable Report Generation**—Enable this option if you do not want to generate a test report.
- **Include Execution Times**—Enable this option if you want to log the time that each step module takes to the report. This includes the time

that the subsequence, LabVIEW VI, or C function takes to execute. It does not include the time that the TestStand engine takes to evaluate preconditions, load the step module, and so on.

- **Include Step Results**—Enable this option if you want to display the results of each step. Disable this checkbox if you want to include only a header for each UUT. The header indicates if the UUT passed or failed.
- **Include Output Values**—Enable this option if you want to log the values that steps acquire to the report. The default process model recognizes specific step properties as containing values to log. These properties include `Result.Numeric`, `Result.String`, and `Result.ButtonHit`. For the Numeric Limit Test built-in step type, `Result.Numeric` contains the numeric measurement that the step acquires. For the String Value Test built-in step type, `Result.String` contains the measurement value that the step returns in string form. For the Message Popup step type, `Result.ButtonHit` contains the number of the button the operator pressed to dismiss the message popup.
- **Include Test Limits**—Enable this option if you want to log values that step types use as test limits to the report. The default process model recognizes specific step properties as containing test limits. These properties include `Limit.Low`, `Limit.High`, `Limit.String`, and `Comp`. The Numeric Limit Test compares the measurement value it acquires against `Limit.Low`, `Limit.High`, or both, and it uses `Comp` to indicate the type of comparison to make. The String Value Test compares the string it acquires against `Limit.String`, and it uses `Comp` to indicate whether to ignore case in the comparison.
- **Append if File Already Exists**—Enable this option if you want to append the report to the target file, if the target file already exists. If you disable this option, the report overwrites the target file. If you create a separate report for each UUT and you disable this option, the report for each UUT overwrites the target file, if it already exists.
- **Report Format**—Use this control to specify the output format of the report file. You can use the menu ring to the right of the control to select either a Web Page format (`.html`) or an ASCII Text format (`.txt`).
- **Result Filtering Expression**—Use this control to select steps to appear in the report. You do so by specifying an expression that the report generator evaluates for each step result. The report generator includes the step in the report if the expression evaluates to `True`.

You can use any subproperty in the `Result` property of the step, but you must you use `%Result` in place of `Step.Result`. For example, if you want to include only failing steps in the report, set the expression to `%Result.Status == "Failed"`. You can use the menu ring to the right of the control to select predefined expressions for all steps, only failing steps, or only passing steps.

- **Report Colors**—Use this control to specify the colors of the report. This option is only available when you select the Web Page format.
- **Select a Report Generator for Producing the Report Body**—Use this option to select between producing the body of the report using sequences or a DLL. The report body is the section of the report between the header and footer that contains individual results for each sequence and step that TestStand called. In the default TestStand process model, the `TestReport` callback makes the determination as to whether the report is built with a sequence or a DLL call.

  If you select the sequence report generation option, `TestReport` calls the `AddReportBody` sequence in either `ReportGen_txt.seq` or `ReportGen_html.seq` to build the report body. The sequence report generator uses a series of sequences with steps that recursively process the result list for the execution.

  If you select the DLL report generation option, `TestReport` calls a function in the `modelsupport.dll` to build the report body. The DLL report generator is a single call into a C DLL that processes the entire result list for the execution before returning. The project and source code for the LabWindows/CVI-built DLL is available. If you select the DLL option, TestStand generates reports faster, but TestStand does not call `ModifyReportEntry` callbacks.

Report File Pathname Tab

Figure 14-4 shows the Report File Pathname tab of the Report Options dialog box.

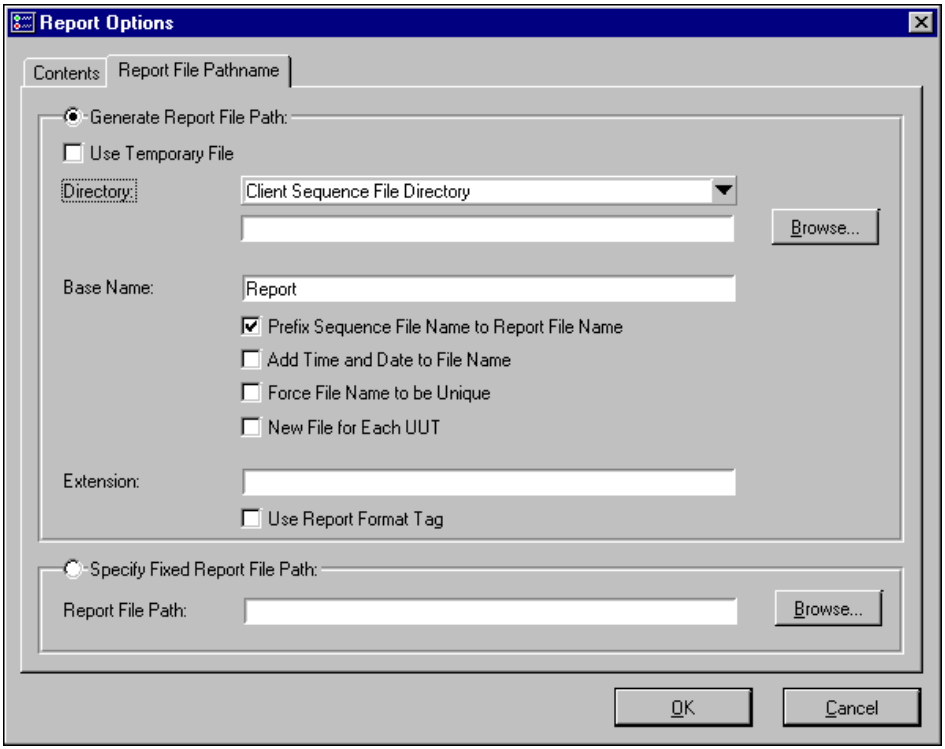

Figure 14-4. Report Options Dialog Box—Report File Pathname Tab

You can specify a fixed pathname to use for all report files, or you can specify options that the report generator uses to generate report file pathnames. The Report File Pathname tab of the Report Options dialog box contains the following controls:

- **Generate Report File Path**—Select this radio button if you want the report generator to create pathnames automatically. When you select Generate Report File Path, you can use the controls in the Generate Report File Path section of the tab.
- **Use Temporary File**—Enable this option if you want to write the report to a temporary file. The report generator deletes the file when you close the Execution window. Enable this option if you do not want to save your test report after you close the Execution window.

- **Directory**—Use these controls to specify the directory in which the report generator writes the report file. In the ring control, you can choose one of the following options.
  - Client Sequence File Directory—The directory that contains the client sequence file. For example if you choose the Test UUTs item from the Execute menu when the `d:\Tests\MySeqs\Seq2.seq` sequence file is active, the report generator writes the report file in the `d:\Tests\MySeqs` directory.
  - <TestStand Directory>\reports\—The `report` subdirectory under the TestStand directory.
  - Specific Directory—A directory you specify in the string control under the ring control. You must enter an absolute path in the string control under the ring control.
- **Base Name**—Use this control to specify the base name for the report filename. Depending on your settings for other options, the report generator might add text to the base name. Do not include a file extension in this control.
- **Prefix Sequence File Name to Report File Name**—Enable this option if you want to add the base name of the client sequence file in front of the name you specify in the Base Name control. For example, if you specify `report` in the Base Name control, the client file name is `auto.seq`, and the report is in HTML format, the report filename is `auto_report.html`.
- **Add Time and Date to File Name**—Enable this option if you want to append a string containing the current time and date in localized format to base name of the report file. For example, `auto_report.html` might become `auto_report[12 47 54 PM][6 24 98].html`.
- **Force File Name to be Unique**—Enable this option if you want to append a unique numeric value to the report file name if the file already exists. For example, `auto_report.html` might become `auto_report_00002.html`.
- **New File for Each UUT**—Enable this option if you want to append the UUT serial number to the report file name. For example, `auto_report.html` might become `auto_report[ABC12345].html`. This causes the report generator to create a separate file for each UUT.

- **Use Report Format Tag**—Enable this option if you want to use the standard file extension for the report format that you specify in the Contents tab. Otherwise, specify a file extension, excluding the dot, in the Extension control.
- **Specify Fixed Report File Path**—Select this radio button if you want to define a specific pathname to use for all report files. You must specify an absolute pathname. Each report file that the report generator creates overwrites the previous report file, unless you enable the Append if File Already Exists option in the Contents tab.

15

Run-Time Operator Interfaces

This chapter gives you an overview of how to create or customize an operator interface application. It also describes the various operator interface applications that TestStand includes.

Overview

TestStand includes three run-time operator interfaces in both source and executable form, so they are fully customizable. Each run-time operator interface is a separate application program that uses the TestStand ActiveX API. The operator interfaces differ primarily based on the language and ADE in which each is developed. TestStand includes run-time operator interfaces developed in LabVIEW, LabWindows/CVI, and Visual Basic. Like the sequence editor, the run-time operator interfaces allow you to start multiple concurrent executions, set breakpoints, and single-step. Unlike the sequence editor, however, the run-time operator interfaces do not allow you to modify sequences, and they do not display sequence variables, sequence parameters, step properties, and so on.

If you are not an experienced programmer, you might find the source code for each run-time operator interface somewhat complex. Before you start attempting to customize the source code for a run-time operator interface, you should first familiarize yourself with the TestStand ActiveX API. To do this, complete the following steps:

1. Thoroughly read the *TestStand ActiveX API Overview* section in the *TestStand ActiveX API Reference* online help. This section contains an overview of the TestStand ActiveX Server functionality and discusses how to call the ActiveX API from different programming languages. Also familiarize yourself with the available methods and properties of each object class in the ActiveX API.
2. Review the example projects and source code located in the `TestStand\Examples\OperatorInterfaces` directory. These examples illustrate the basic programming requirements for creating a simple operator interface application that uses the TestStand ActiveX API.

The first decision you need to make is whether you should customize one of the run-time operator interfaces that TestStand includes, or create your own application from the ground up. For example, you might want a simple operator interface application on your production floor that does not allow you to debug an execution or display the details of an execution the TestStand engine is running. Attempting to customize and remove functionality from a fully functional run-time operator interface application might be more work than is necessary. Instead, you can customize one of the examples in the `TestStand\Examples\OperatorInterfaces` directory or create your own application from the ground up.

TestStand Run-Time Operator Interfaces

TestStand installs the executable, project, and source files for each fully functional run-time operator interface in the `TestStand\OperatorInterfaces\NI` directory tree. If you want to customize one of these run-time operator interfaces, copy the operator interface project and source files from the `NI` subdirectory to the `TestStand\OperatorInterfaces\User` subdirectory before customizing them. This ensures that you do not lose your customizations when you install newer versions of TestStand. In addition, National Instruments recommends that you track the changes you make to the operator interface source so that you can add any future enhancements from newer versions of the TestStand run-time operator interfaces.

The LabWindows/CVI Run-Time Operator Interface

TestStand installs the executable, project, and source files for the LabWindows/CVI run-time operator interface in the `TestStand\OperatorInterfaces\NI\CVI` directory. Table 15-1 lists the files included in the `testexec.prj` project file and describes the purpose of each file.

Table 15-1. Files in the LabWindows/CVI Run-Time Operator Interface Project File

File	Description
`cfgfile.c`	Contains code to save and restore settings, and the most-recently-used files list to a file in the same directory as the executable or project.
`cvibmp.c`	Contains code to translate icon bitmaps from the Windows bitmap format into the LabWindows/CVI bitmap format.

Table 15-1. Files in the LabWindows/CVI Run-Time Operator Interface Project File (Continued)

File	Description
`data.c`	Contains global settings and data that other source modules access Contains lists of data about loaded sequence files, executions, icons, and adapters. It also contains an API to access the lists of data.
`engine.c`	Contains all the code that accesses the TestStand ActiveX automation server. Also creates and destroys the records of data for sequence files, executions, sequences, steps, and so on.
`exedisp.c`	Contains all the code for updating execution displays. Each execution display has its own data record for its panel. Many of the functions in this module access that data record to update settings, data, and display information.
`filelist.c`	Contains code to maintain, save, and restore the most-recently-used file list at the bottom of the **File** menu.
`main.c`	Contains the main procedure for the program, and consequently calls the initialization and cleanup routines for the application. It also contains the highest-level code for processing the command line arguments.
`maingui.c`	Contains all the graphical user interface code that is not specific to the execution display or sequence display. This includes code to handle the single window (tab-dialog) setting of the application as well as initialization and cleanup of the different display components such as the **Tools** menu. Also, all user interface callbacks, which are common to both sequence displays and execution displays, are located in this file. For example, menu item callbacks that are common to both sequence and execution displays.
`seqdisp.c`	Contains code for updating the sequence display, where you can launch executions and load and display sequences. The application uses only one sequence display at a time.
`teerror.c`, `teerror.h`	Contains code to report and display error messages. The header file provides several useful error-checking macros.
`rnstchng.c`	Contains run-state change callbacks used to control the flow of UIMessages from the TestStand engine when suspended at a breakpoint in the source code for the Operator Interface. Also contains code to cleanup properly when terminating the operator interface prematurely from within LabWindows/CVI.
`tsapicvi.fp`	The TestStand ActiveX API wrapper functions.

Refer to the file, `TestStand\OperatorInterfaces\NI\CVI\readme.doc`, for any additional information about the LabWindows/CVI run-time operator interface project.

The LabVIEW Run-Time Operator Interface

TestStand installs the executable and source files for the LabVIEW operator interface in the `TestStand\OperatorInterfaces\NI\LV` directory. Table 15-2 shows the three top-level VIs in the LabVIEW Run-Time Operator Interface.

Table 15-2. Top-Level Files in the LabVIEW Run-Time Operator Interface

File	Description
`TestStand - Runtime Operator Interface.vi`	This VI launches the operator interface by creating a reference to the TestStand ActiveX automation server and dynamically loads and calls `TestStand - Sequence Display.vi`.
`TestStand - Sequence Display.vi`	This VI displays the Sequence Display window of the operator interface. Whenever a new execution starts, the hierarchy of the Sequence Display creates a new instance of `TestStand - Execution Display.vi`.
`TestStand - Execution Display.vi`	This VI is the master VI for the Execution Display window of the operator interface. For every new execution started, with the exception of executions started during the shutdown procedure, the `TestStand - Sequence Display.vi` hierarchy makes a temporary copy of this VI and runs it. Depending on whether the execution starts hidden or not, this VI also opens its own panel.

Refer to the file, `TestStand\OperatorInterfaces\NI\LV\readme.doc`, for any additional information about the LabVIEW run-time operator interface VIs.

Building a Standalone Executable

Use the following steps to make an executable version of the LabVIEW run-time operator interface.

Note ***You must have the LabVIEW Application Builder to perform these steps.***

1. Open `TestStand - Runtime Operator Interface.vi` in LabVIEW.
2. Enable the Run When Opened option under the Execution Options section of the VI Setup dialog box.

3. Save the VI using the **Save with Options** command in the **File** menu and select the Application Distribution option. Save the VI into a new VI library with the name `maingui.llb` in some directory of your choosing.
4. Click on **File»Edit Library** to make this VI the top-level VI in `maingui.llb`.
5. Click on **Project»Build Application** to build the operator interface application. Embed the `maingui.llb` library file in the application.
6. Enable the ActiveX server option and enter a unique name in the ProgID Prefix control, such as `TestStandLVGUI`.
7. Name the resulting executable `testexec.exe`.

Note ***If you enable the ActiveX server, you can configure the TestStand engine to use the LabVIEW run-time embedded in the executable as the LabVIEW server that runs VI tests. Refer to the Configuring the LabVIEW Standard Prototype Adapter section in Chapter 12, Module Adapters, for more information.***

8. Open the VI `TestStand\Sequence Display\TestStand - Sequence Display.vi` and save the VI with the Application Distribution option into a new library file, `testexec.llb`, in the same directory as `testexec.exe`.
9. Open `TestStand\Execution Display\TestStand - Execution Display.vi` and save the VI with the Application Distribution option into a new library file, `testexec.llb`, in the same directory as `testexec.exe`.
10. Open `TestStand\Common VIs\TestStand - Busy Indicator.vi` and save the VI with the Application Distribution option into a new library file, `testexec.llb`, in the same directory as `testexec.exe`.

You can now run the operator interface by launching `testexec.exe`.

The Visual Basic Run-Time Operator Interface

TestStand installs the executable, project, and source files for the Visual Basic operator interface in the `TestStand\OperatorInterfaces\NI\VB` directory. Table 15-3 shows the three top-level VIs in the Visual Basic run-time operator interface.

Table 15-3. Top-Level Files in the Visual Basic Run-Time Operator Interface

Files	Description
Forms	
`AdapterCfg.frm`	An implementation of an adapter configuration dialog box that calls the internal adapter configuration dialog box of the adapter that the user selects.
`DoNothing.frm`	A dialog that immediately unloads itself. Use this dialog box to make Visual Basic remove any menus that are displayed when the menus need to be dynamically updated because of a change in the execution state of a sequence that TestStand is running.
`ExeDisplay.frm`	The code that implements the execution displays. All the callbacks and source code which relate to maintaining and updating an execution display are contained within this file.
`OkBox.frm`	A simple text message dialog box that contains a scrollable text control. Use this dialog box to report error messages.
`SeqDisplay.frm`	The code that creates the sequence display window. This file contains all the callbacks and source code that relate to maintaining and updating the sequence display, as well as code to start executions.
`Splash.frm`	The about dialog box.
`TermAbortCancel.frm`	A dialog box that gives the user the choice of terminating, aborting, or canceling an execution. This dialog box appears when a user attempts to close an execution display of an execution that has not finished running.

Table 15-3. Top-Level Files in the Visual Basic Run-Time Operator Interface (Continued)

Files	Description
Modules	
`Data.bas`	Contains global settings and data that other source modules access. This module is also responsible for initialization and cleanup of the `LoadedSeqFileList.bas` module and the `ExeList.bas` module.
`ErrorHandler.bas`	Contains code for displaying the current error information contained in the Visual Basic `global Err` object.
`ExeList.bas`	Contains code for maintaining a list of execution displays and their corresponding executions. Also, provides methods and properties to perform different operations on them and get information about them.
`LoadedSeqFileList.bas`	Maintains the list of the sequence files that are loaded for the sequence display. The `SeqDisplay.frm` calls functions in this module to load and unload sequence files and get information about the list.
`MiscGUI.bas`	Contains utility functions used by both `SeqDisplay.frm` and `ExeDisplay.frm`. Also, contains code to start the login/logout callback and to maintain the **Tools** menu items and entry point menu items for the displays.
Class Modules	
`EntryPointMenu.cls`	Contains code for maintaining and updating the menus created for the entry points of a model sequence file. Instances of this class are created for every menu that can contain entry points.
`WaitCursor.cls`	Contains code for displaying a wait cursor for the life of an object created as an instance of this class. When the object is terminated, it resets the cursor back to its previous state.

Refer to the file, `TestStand\OperatorInterfaces\NI\VB\readme.doc`, for more information about the Visual Basic run-time operator interface project.

Distributing a Run-Time Operator Interface

Refer to Chapter 16, *Distributing TestStand*, for more information about distributing the TestStand engine with your customized run-time operator interface application.

16

Distributing TestStand

This chapter describes how to create an installer for a customized TestStand engine, how to distribute the TestStand engine with a run-time operator interface, and how to distribute each type of code module that TestStand supports. This chapter also describes how to customize and distribute a LabVIEW run-time server.

Creating a Run-Time TestStand Engine Installation

If you want to distribute a run-time version of the TestStand engine with your operator interface application, you must create a separate installer for the TestStand engine. When you distribute your operator interface application, you can either install the TestStand engine separately or customize the operator interface installer to call the installer for your TestStand engine.

TestStand includes a wizard for creating a custom TestStand engine installation. Complete the following steps to create a custom TestStand engine installation.

1. Select **Installation Wizard for the TestStand Engine** from the TestStand program group to launch the wizard.

   Figure 16-1 shows the opening dialog box for the wizard.

Figure 16-1. Opening Dialog Box for the TestStand Engine Installation Wizard

2. Click on the **Begin** button.

   The wizard displays the dialog box as shown in Figure 16-2. This dialog box lists the additional files the wizard includes in the installation for the TestStand engine. By default, the wizard includes the `TestStand\Components\User` directory in the custom installer, which ensures that the installer contains any custom engine components you create. In addition, the wizard adds the `ToolMenu.ini` file from your TestStand station. Refer to Chapter 3, *Configuring and Customizing TestStand*, for more information about the TestStand components you can customize.

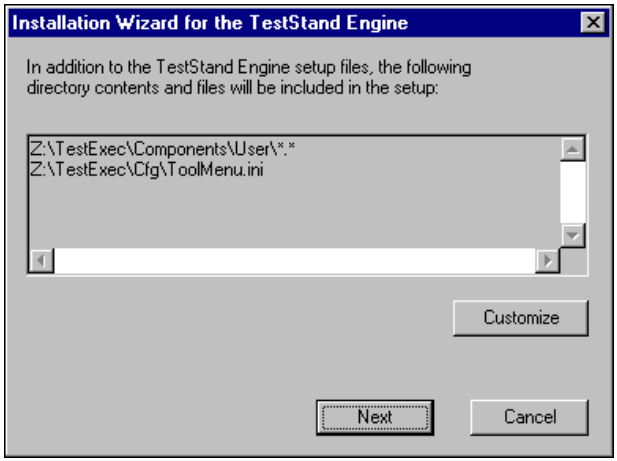

Figure 16-2. Default Components to Include in the Installation

3. Click on the **Customize** button to select which additional files the wizard includes in the installation. When you make this selection, the wizard displays the Customize Files to Include in Installation dialog box, as shown in Figure 16-3.

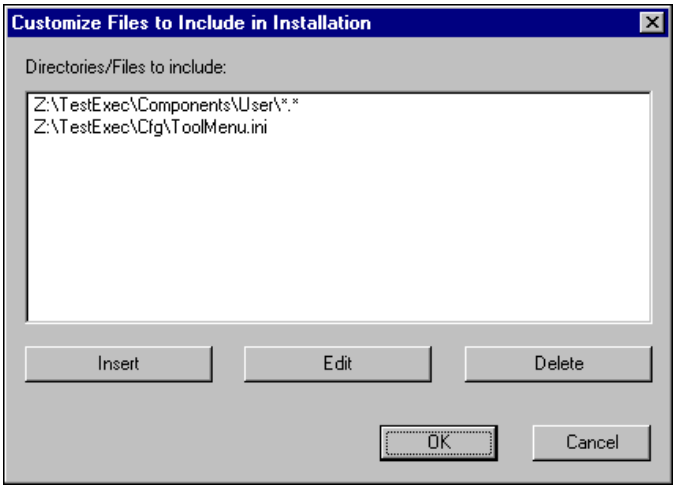

Figure 16-3. Customize Files to Include in Installation Dialog Box

4. Click on the **Insert** button to insert new entries in the file list or click on the **Edit** and **Delete** buttons to edit and delete existing entries. When you insert or edit an entry, the wizard displays the Select Files to Include dialog box, as shown in Figure 16-4.

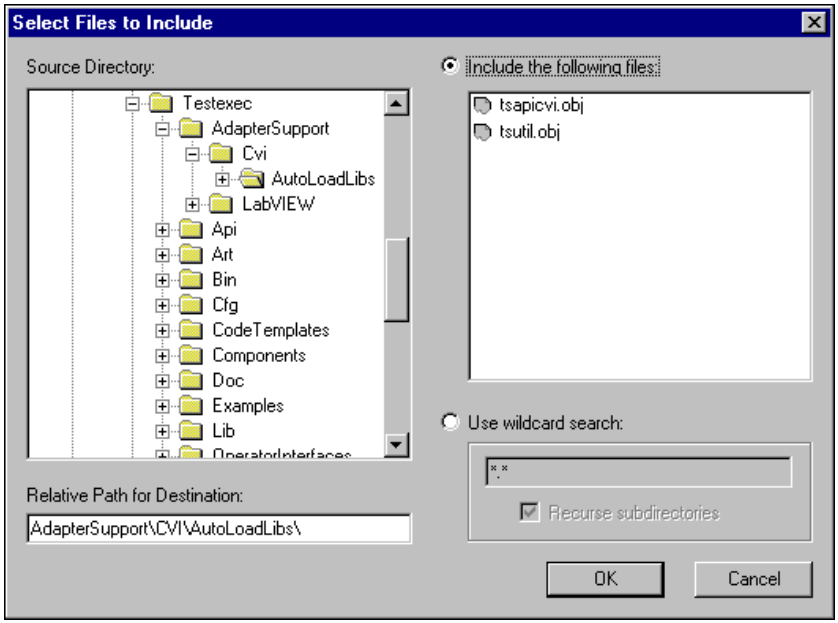

Figure 16-4. Select Files to Include Dialog Box

5. Include individual files or include all files that match a filename containing wildcard characters. When you specify files using wildcard characters, you can recurse subdirectories, and the resulting installation maintains the directory structure when distributing these files to a target system. You use the Relative Path for Destination control to specify the destination subdirectory where the installation installs the specified files under the distributed TestStand engine.

6. After you define the list of additional files to include in the installation, the wizard prompts you for the directory you want the wizard to create the installation files in. Upon completing the build process, the wizard creates the following installation files:

   - `TSEngine.cab`—Compressed file that contains the TestStand engine files.
   - `SetupTSEngine.exe`—Setup executable that uncompresses and installs the TestStand engine.

Using a Custom TestStand Engine Installation

You can invoke the custom TestStand engine installation by running the setup executable file separately or by calling it from another installation. The setup executable supports the following command-line options:

`-x` Delete the `TSEngine.cab` file after installing the TestStand engine, and register with the operating system to delete the `SetupTSEngine.exe` file after rebooting the target system.

`-noprompt` Do not prompt.

`<path>` Install at specified location, default is `C:\TestStand.`

Table 16-1 lists the actions the installer takes depending on whether the TestStand engine is already installed on a target system and which command-line options you pass to the setup executable.

Table 16-1. Custom TestStand Engine Installer Actions

Engine Already Installed?	-noprompt Specified?	<path> Specified?	Installer Actions
No	No	No	Prompt to specify install directory. Installer uses `c:\TestStand` as default.
No	No	Yes	Prompt to specify install directory. Installer uses command-line specified path as default.
No	Yes	No	No prompt for installation directory. Installer uses `c:\TestStand` as the install directory.
No	Yes	Yes	No prompt for installation directory. Installer uses command-line specified path as default.
Yes	No	Yes/No	Prompt to confirm installation. Installer uses previously installed location.
Yes	Yes	Yes/No	No prompt. Installer uses previously installed location.

Refer to the *Distributing your Operator Interface* section later in this chapter for recommendations on how to bundle a custom engine installation with a distribution of your operator interface application.

Distributing your Operator Interface

Installing the Customized Engine

The following sections explain how to bundle a custom TestStand engine installation with your distribution kit for LabVIEW, LabWindows/CVI, and Visual Basic.

LabVIEW

You can use the Create Distribution Kit feature in the LabVIEW development environment to create an installation for your operator interface. If you want to bundle a custom TestStand engine installation in your LabVIEW distribution kit you must complete the following steps:

1. Add the `SetupTSEngine.exe` and `TSEngine.cab` files to your distribution kit. You can install the file in the base installation directory of your application.
2. In the Advanced section of the Create Distribution Kit dialog box, select the `SetupTSEngine.exe` file in the Executable to Run After Setup section. Specify the `-x` command-line option to delete the engine installation files after the executable runs.
3. If you want to alter the default message at the end of the installation of your application to indicate that the TestStand engine will install next, you can use a custom template file as the installation script. TestStand includes a custom script file, `TestStand\OperatorInterfaces\LV\TestStandLVTemplate.inf`, which is based on the default LabVIEW 5.0.1 template file `LabVIEW\APPLIBS\distkit\template.inf`. The custom script contains an altered `ExitSuccess` procedure.

Refer to the LabVIEW documentation for more information on using the advanced options of the Create Distribution Kit feature in LabVIEW.

LabWindows/CVI

You can use the Create Distribution Kit feature in the LabWindows/CVI development environment to create an installation for your operator interface. If you want to bundle a custom TestStand engine installation in your LabWindows/CVI distribution kit you must complete the following steps:

1. Add the `SetupTSEngine.exe` and `TSEngine.cab` files to your distribution kit. You can install the file in the base installation directory of your application.

2. On the Advanced Distribution Kit Options dialog box, select the `SetupTSEngine.exe` file in the Executable to Run After Setup section. Specify the `-x` command-line option to delete the engine installation files after the executable runs.
3. If you want to alter the default message at the end of the installation of your application to indicate that the TestStand engine will install next, you can use a custom template file as the installation script. TestStand includes a custom script file, `TestStand\OperatorInterfaces\CVI\TestStandCVITemplate.inf`, which is based on the default LabWindows/CVI 5.0.1 template file `CVI\bin\template.inf`. The custom script contains an altered `ExitSuccess` procedure.

Refer to the LabWindows/CVI documentation for more information on using the Advanced Distribution Kit dialog box of the Create Distribution Kit feature in LabWindows/CVI.

Visual Basic

You can use the Application Setup Wizard feature in Visual Basic to create an installation for your operator interface. If you want to bundle a custom TestStand engine installation in your Visual Basic 5.0 application installation, you must complete the following steps:

1. Update the `VisualBasic\SetupKit\Setup1\Setup1.vpb` project to automatically launch `SetupTSEngine.exe` after successfully installing your operator interface application. You can do this by inserting code into the `Form_Load` subroutine in the `Setup1.frm` module. You might want to review existing code in the `Setup1.vpb` project that calls the `AXDIST.EXE` and `WINT351.EXE` installers dynamically when you include the files in the application installation.

   You cannot use the `FsyncShell` function to launch the TestStand engine installer. The `FsyncShell` function prevents the TestStand engine installer from running properly. If you want to wait for the TestStand installer to complete its installation before completing the application installation, you can use the `ShellAndWait` function in the ShellAndWait module that TestStand includes in the `TestStand\OperatorInterfaces\VB` directory.

   If you want to automatically delete the engine installation files after the executable runs, you can specify the `-x` command-line option when calling the TestStand engine installer.
2. Add the `SetupTSEngine.exe` and `TSEngine.cab` files to your installation. You can install the file in the installation directory of your application.

Refer to the Visual Basic documentation for more information on using the Application Setup Wizard feature in Visual Basic.

Distributing Sequences and Code Modules

This section explains how to distribute sequence files, DLL code modules, object code modules, static library code modules, LabVIEW test VIs, and ActiveX automation code modules.

Distributing Sequence Files

For each step in a sequence that calls a code module, TestStand stores the module name and path as properties of the step. The path can be an absolute path or a path that is relative to a directory in the TestStand search directories. When you distribute a sequence file, you also must distribute the appropriate step modules and their support files onto the target system. Also, you must ensure that sequence files can locate their step module files using the TestStand search paths list.

If you distribute a sequence file that contains absolute paths, TestStand will not find its code modules unless the target system contains a similar directory structure. National Instruments recommends that you use relative paths whenever possible. You can modify the precedence of the directory paths that TestStand searches with the **Configure»Search Directories** command.

Distributing DLL Code Modules

A DLL file can require that other support DLL files be installed on a system so TestStand can properly load the DLL into memory. You must ensure that you install the appropriate support DLL files on a target system before running the DLL tests within TestStand.

Distributing Object and Static Library Code Modules

When the C/CVI Standard Prototype Adapter loads an object or static library file, the LabWindows/CVI Run-time Engine resolves all external references in the file. When you distribute object or static library code modules, you must distribute the appropriate support files to the target system.

When running object or static library code module tests in-process, the adapter must load the support libraries that the code module depends on before it loads the code module file. The adapter automatically loads all support libraries from the

`TestStand\AdapterSupport\CVI\AutoLoadLibs` directory. You must ensure that you copy the appropriate support files to the parallel directory on a target system. One option is for you to include the contents of the `AutoLoadLibs` directory on your development system in the distribution of the custom TestStand engine.

If you want a TestStand step to call a code module out-of-process in an external instance of LabWindows/CVI, you must include all support libraries other than LabWindows/CVI libraries in the project on the target system.

Refer to the *Configuring the C/CVI Standard Prototype Adapter* section in Chapter 12, *Module Adapters*, for more information about using different code modules with the C/CVI Standard Prototype Adapter.

Distributing LabVIEW Test VIs

The LabVIEW Standard Prototype Adapter loads and runs VIs using a LabVIEW ActiveX server. The LabVIEW server can be the LabVIEW development environment or a LabVIEW-built run-time application that includes the LabVIEW ActiveX server. When you distribute a TestStand test VI, you must ensure that the LabVIEW server can locate all subVIs. The method you use to guarantee that the LabVIEW server can locate subVIs depends on how you want to distribute your source VIs.

When you develop your test VI in LabVIEW, you usually save the VI without its hierarchy. For each subVI reference in a VI, LabVIEW saves the location of the subVI within the VI. When TestStand requests a LabVIEW server to load a VI, the server attempts to locate all subVIs in the VI hierarchy. If the LabVIEW server cannot find the subVI in the expected location that is stored in the VI, the LabVIEW server searches the VI search path list as defined in the preferences for the server.

A LabVIEW server reads its search path list from a `.ini` file with the same base name as the server application, that is, `LabVIEW.ini` or `TestStandLVRTS.ini`. By default, the search path preferences for a LabVIEW server are as follows:

1. The directory that contains the top-level VI being opened.
2. The list of directories that the LabVIEW server builds each time a VI is loaded.
3. The `vi.lib` subdirectory in the Library directory for the LabVIEW server.

4. The `user.lib` subdirectory in the Library directory for the LabVIEW server
5. The `instr.lib` subdirectory in the Library directory for the LabVIEW server.

Refer to the *Search Paths* topic in the *LabVIEW Online Reference* for more information about the VI Search Path preference.

The rest of this section describes three options for distributing your VIs. You might want to use one of these options or a combination of these options.

Packaging VIs and SubVIs for a Sequence File

TestStand includes a **Tools** menu utility that can help you save the entire test VI hierarchy for a specific sequence file. For all steps in a sequence file that use the LabVIEW Standard Prototype Adapter, the utility can save test VIs to a single directory and all subVIs, run-time menu files, and external subroutines to a separate VI library. You can remove the diagrams from all the VIs. You can run this utility by clicking on **Tools»Assemble Test VIs for Run-Time Distribution** while a sequence file window is active.

When you create a run-time distribution kit you must install the VIs and support VI library that the utility creates on your target system. Also, you must ensure that sequence files that call the VI tests can locate the files using the TestStand search paths list.

If your tests call any subVIs dynamically, the packaging utility does not save the subVIs in the support VI library. You must distribute these dynamically-called VIs separately.

Distributing VIs by Saving Them without Full Hierarchy

If you want to maintain your test VIs on a target system as independent files, and you do not want to resave your VI libraries with their full hierarchy, you must distribute all required subVIs and support files to the target system. Support files include external subroutines, run-time menus, and DLLs. This includes distributing VIs and VI libraries from the LabVIEW library subdirectories, that is, `vi.lib`, `user.lib`, and `instr.lib`, and any other files from additional directories in your search path preferences. In addition, if you want to maintain multiple LabVIEW servers on your target system, you must ensure that each LabVIEW server can find any required subVIs.

For example, if your development system contained a directory structure of sequences and VIs, you could distribute your VIs as follows:

1. Duplicate the entire directory structure of sequences and VIs on your target system. If you do not install the files in the same absolute path, you must make sure the sequences and VIs contain relative paths for references to other files within this directory structure.
2. Copy the required subVIs and VI libraries from the library subdirectories of the LabVIEW development system to the appropriate library subdirectories of each LabVIEW server. If the target system already contains a copy of the LabVIEW development environment, you need only copy additional files that the target system does not have. If the target system contains only a LabVIEW run-time server, you can copy the entire library subdirectories from the development system to the target system. If the target system contains multiple LabVIEW servers, you can maintain a single LabVIEW server library directory by customizing the preferences for each server to reference to this single library.

If you upgrade the version of LabVIEW on your systems, you must rebuild all LabVIEW run-time servers with the new version of LabVIEW, mass compile your test VIs and subVIs, and update your library subdirectories where appropriate.

Distributing VIs by Saving Them with Full Hierarchy

LabVIEW allows you to save your VIs with their full hierarchy into a directory or a VI library. This includes saving all subVIs, controls, and external subroutines, including the ones in `vi.lib`. You can remove the diagrams from all of the VIs.

Using this LabVIEW feature, you can resave your test VIs with their full hierarchy to a new, separate directory image that you can distribute to a target system. If your sequence refers to test VIs from within VI libraries, you must save the VI under the same VI library name so that the pathname for the VI module in the sequence is correct. If your sequence refers to test VIs directly from disk, you cannot save the VI hierarchy within a VI library.

To save the full hierarchy for a VI, select **File»Save with Options»Custom Save** with the following options selected.

- To new location - single prompt.
- Save entire hierarchy.
- Include `vi.lib` files (This selection is necessary only if the target LabVIEW server is not the LabVIEW development environment.)

- Include external subroutines.
- Include run-time menus.
- Remove diagrams (This selection is optional.).

To streamline the saving process, you can create a VI that contains all test VIs on its diagram. Then save this VI and its VI hierarchy. When you do this, you do not have to save VIs individually for every VI used with a sequence. Instead, you perform the saving procedure only once.

If your tests call any subVIs dynamically, you must distribute these dynamically-called VIs separately.

Distributing ActiveX Automation Code Modules

When the ActiveX Automation Adapter attempts to load an ActiveX Automation server, the server must be registered with the operating system. When distributing TestStand, you must ensure that you properly install and register ActiveX Automation server code modules on a target system before using the server from within TestStand.

Customizing and Distributing a LabVIEW Run-Time Server

The LabVIEW Standard Prototype Adapter runs VIs using a LabVIEW ActiveX server. The server can be the LabVIEW development environment or a LabVIEW-built run-time application that includes the LabVIEW ActiveX server. TestStand requires a LabVIEW run-time system when you do not install LabVIEW on the system and you run sequences that contain steps calling LabVIEW VIs.

The TestStand installation includes a prebuilt LabVIEW run-time system with source in the `TestStand\Components\NI\RuntimeServers\LabVIEW` directory tree. The executable name is `TestStandLVRTS.exe` and the ActiveX server name is `TestStandLVRTS`. If you want to customize the server, copy all source files from the `NI` subdirectory to the `TestStand\Components\User\RuntimeServers\LabVIEW` subdirectory before customizing them. This ensures that you do not lose your customizations when you install newer versions of TestStand.

Refer to the *Configuring the LabVIEW Standard Prototype Adapter* section in Chapter 12, *Module Adapters*, for more information about configuring which LabVIEW server TestStand uses.

Note ***When running the application version of the LabVIEW run-time operator interface, you can use the ActiveX server from the operator interface application instead of the server from the LabVIEW development environment or the prebuilt LabVIEW run-time system.***

Rebuilding the TestStand LabVIEW Run-Time Server

The prebuilt LabVIEW run-time server is built with a specific version of LabVIEW. Refer to the `readme.txt` file in the same directory as the executable for the specific version. Whenever you save your VIs with a newer version of LabVIEW, you must rebuild any LabVIEW run-time servers that TestStand uses to execute the newer VIs. To rebuild or customize the TestStand LabVIEW run-time server, you must complete the following steps:

1. Copy all source files, with the exception of `TestStandLVRTS.exe`, from the `TestStand\Components\NI\RuntimeServers\LabVIEW` subdirectory to the `TestStand\Components\User\RuntimeServers\LabVIEW` subdirectory.
2. Open `Server.llb\TestStand - LabVIEW Runtime Server.vi` in LabVIEW.
3. Select **VI Setup»Execution Options** and enable the Run When Opened option.
4. Select **File»Save with Options** and save the VI for Application Distribution to a new VI library, `TestStandLVRTS.llb` in the `TestStand\Components\User\RuntimeServers\LabVIEW` directory.
5. Select **File»Edit VI Library** and make `TestStand - LabVIEW Runtime Server.vi` the top-level VI in `TestStandLVRTS.llb`.
6. Select **Project»Build Application** to build an executable with the following settings:
   - Embedded Library—`TestStandLVRTS.llb`
   - ActiveX server—Enabled
   - ProgID Prefix—`TestStandLVRTS`
   - Application Name—`TestStandLVRTS.exe`
   - Application Directory—`TestStand\Components\User\RuntimeServers\LabVIEW`

Distributing the TestStand LabVIEW Run-Time Server

The TestStand Engine Installation Wizard automatically includes the default NI LabVIEW run-time server with any engine installation. If you include the `TestStand\Components\User` directory in the custom engine installation, any customized version of the LabVIEW run-time server is also included in the custom engine installation. The resulting engine installation automatically registers the `NI` LabVIEW run-time server first and then the `User` LabVIEW run-time server. If the `User` version uses the same ProgID, its registration replaces the previously registered `NI` server.

To manually distribute the LabVIEW run-time server, you must include the following files:

- `TestStandLVRTS.exe`
- `TestStandLVRTS.tlb`
- `LVWUtil32.dll`

If you want to manually register the ActiveX server in a LabVIEW run-time application, you can launch the executable with the `/Register` keyword on the command-line. When you do this, the executable registers itself and terminates. You can also register the server by simply launching the executable.

You also must distribute any required files to load and run a test VI dynamically. If your application uses serial port or data acquisition functionality, you must include the `serpdrv` or `daqdrv` files from the LabVIEW development system in the same directory as the executable file. If your application uses a GPIB or data acquisition board, you must install the hardware drivers that come with the board.

If you choose to distribute your test VIs to a target system as independent files, and you do not want to resave your VI libraries with their full hierarchy, you must also distribute any files required by the test VIs, that is, files from the `vi.lib`, `user.lib`, and `instr.lib` directories. Refer to the *Distributing VIs by Saving Them without Full Hierarchy* section in this chapter for more information.

Refer to the *LabVIEW Application Builder Release Notes* documentation for more information about creating a LabVIEW application that includes the LabVIEW ActiveX server.

A

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call 512 795 6990. You can access these services at:

United States: 512 794 5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at 512 418 1111.

E-Mail Support (Currently USA Only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

`support@natinst.com`

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

Country	Telephone	Fax
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	011 288 3336	011 288 8528
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Québec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 6120092	03 6120095
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___) _____Phone (___) _____

Computer brand_____ Model _____Processor_____

Operating system (include version number) _____

Clock speed _____MHz RAM _____MB Display adapter _____

Mouse ___yes ___no Other adapters installed_____

Hard disk capacity _____MB Brand_____

Instruments used _____

_____

National Instruments hardware product model _____ Revision _____

Configuration _____

National Instruments software product _____ Version _____

Configuration _____

The problem is: _____

_____

_____

_____

_____

List any error messages: _____

_____

_____

The following steps reproduce the problem: _____

_____

_____

_____

_____

TestStand Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

National Instruments Products

Hardware revision _____

Interrupt level of hardware _____

DMA channels of hardware _____

Base I/O address of hardware _____

Programming choice _____

National Instruments software _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Other Products

Computer make and model _____

Microprocessor _____

Clock frequency or speed _____

Type of video board installed _____

Operating system version _____

Operating system mode _____

Programming language _____

Programming language version _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *TestStand User Manual*

Edition Date: December 1998

Part Number: 322016A-01

Please comment on the completeness, clarity, and organization of the manual.

_____

_____

_____

_____

_____

_____

_____

If you find errors in the manual, please record the page numbers and describe the errors.

_____

_____

_____

_____

_____

_____

_____

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

E-Mail Address _____

Phone (___) _____ Fax (___) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, Texas 78730-5039

Fax to: Technical Publications
National Instruments Corporation
512 794 5678

Glossary

Prefix	Meaning	Value
p-	pico	10^{-12}
n-	nano-	10^{-9}
µ-	micro-	10^{-6}
m-	milli-	10^{-3}
k-	kilo-	10^{3}
M-	mega-	10^{6}
G-	giga-	10^{9}
t-	tera-	10^{12}

A

abort — To stop an execution without running any of the Cleanup step groups in the sequences on the call stack run. When you abort an execution, no report generation occurs.

active window — The window that user input affects at a given moment. The title of an active window is highlighted.

ActiveX reference property — A container of information that maintains a reference to an ActiveX object. TestStand maintains the value of the property as an `IDispatch` or `IUnknown` pointer.

ActiveX server — Any executable code that makes itself available to other applications according to the ActiveX standard. ActiveX implies a client/server relationship in which the client requests objects from the server and asks the server to perform actions on the objects.

Adapter — A service of the TestStand engine that steps use to invoke code in another sequence or in a code module. The adapter knows the type of the code module, how to call it, and how to pass parameters to it.

administrator — A user profile that usually contains all privileges for a test station.

Application Development Environment (ADE)	A programming environment such as LabVIEW, LabWindows/CVI, or Microsoft Visual C, in which you can create test modules and run-time execution operator interfaces.
Application Programming Interface (API)	A set of classes, methods, and properties that you use to control a specific service, such as the TestStand engine.
array property	A property that contains an array of single-valued properties of the same type.
ASCII	American Standard Code for Information Interchange.

B

block diagram	Pictorial description or representation of a program or algorithm. In LabVIEW, the block diagram which consists of executable icons called nodes and wires that carry data between the nodes, is the source code for the VI. The block diagram resides in the Diagram window of the VI.
breakpoint	An interruption in the execution of a program.
built-in property	A property that all steps or sequences contain. An example is the step run mode property. TestStand normally hides these properties in the sequence editor, although it lets you modify some of them through dialog boxes.
built-in step type property	A property that is common to all steps of the same type. A built-in step type property is either a class step type property or an instance step type property.
button	A dialog box item that, when selected, executes a command associated with the dialog box.

C

checkbox	A dialog box input that allows you to toggle between two possible options.
cluster	A set of ordered, unindexed data elements in LabVIEW of any data type including numeric, Boolean, string, array, or cluster. The elements must be all controls or all indicators.

class — Defines a list of methods and properties that you can use with respect to the objects that you create as instances of that class. A class is like a data type definition except that it applies to objects rather than variables.

class step type property — A built-in step property that exists only in the step type itself. TestStand uses these properties to define how the step type works for all step instances. Step instances do not contain their own copies of class properties.

client sequence file — A sequence file that contains the main sequence a process model invokes to test a UUT. Each client sequence file contains a sequence called `MainSequence`. The process model defines what is constant about your testing process, whereas the client sequence file defines the steps that are unique to the different types of tests you run.

clipboard — A temporary storage area the operating system uses to hold text that is cut, copied, or deleted from a work area.

code module — A program module, such as a Windows Dynamic Link Library (`.dll`) or LabVIEW VI (`.vi`), that contains one or more functions that perform a specific test or other action.

code template — A source file that contains skeleton code. The skeleton code serves as a starting point for the development of code modules for steps that use the step type.

configuration entry point — A sequence in the process model file that configures a feature of the process model. Configuration entry points usually save configuration information in a `.ini` file in the `TestStand\cfg` directory. By default, configuration entry points appear in the Configure menu. For example, the default process model contains the configuration entry point: `Config Report Options`. The `Config Report Options` entry point appears as **Report Options** in the **Configure** menu.

connector — Part of a LabVIEW VI or function node that contains its input and output terminals, through which data passes to and from the node.

container property — A property that contains no values, and typically contain multiple subproperties. Container properties are analogous to structures in C/C++ and to clusters in LabVIEW.

context menu — Menus accessed by clicking on an object. Menu options pertain to that object specifically.

control — An input and output device for entering data that appears on a panel or window.

control flow — The sequential order of instructions that determines execution order.

custom named data type — A data type that you define and name. For example, you might create a `Transmitter` data type that contains subproperties such as `NumChannels` and `PowerLevel`.

custom property — A property that you define in a step type. Each step you create with the step type has its own copy of the custom property. TestStand uses the value that you specify for the custom property in the step type as the initial value of the property in each new step you create. Normally, after you create a step, you can change the value of the property in the step.

D

dialog box — A prompt mechanism in which you specify additional information needed to complete a command.

developer — A user profile that usually contains all privileges associated with operating, debugging, and developing sequences and sequence files, but cannot configure user privileges, report options, or database options.

DLL — dynamic link library

E

Edit substep — A substep that the engine calls when editing the step. You invoke the substep with the menu item that appears in the context menu above **Specify Module**. The Edit substep displays a dialog box in which the sequence developer edits the values of custom step properties. For example, the Edit Limits item appears in the context menu for Numeric Limit test steps, and the Edit Pass/Fail Source item appears in the context menu for Pass/Fail test steps.

engine — *See* Test Executive Engine

engine callback — A sequence that TestStand invokes at specific points during execution. You use engine callbacks to tell TestStand to 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.

entry points — A sequence in the process model file that TestStand displays as a menu item, such as `Test UUTs`, `Single Pass`, and `Report Options`.

error occurred flag — A Boolean flag, `Step.Result.Error.Occurred`, that indicates whether a run-time error occurred in the step.

execution — An object that contains all the information TestStand needs to run a sequence, its steps, and any subsequences it calls. Typically, the TestStand sequence editor creates a new window for each execution.

execution entry point — A sequence in a process model that runs tests against a UUT. Execution entry points call the `MainSequence` callback in the client sequence file. The default process model contains two execution entry points: `Test UUTs` and `Single Pass`. By default, execution entry points appear in the Execute menu. Execution entry points appear in the menu only when the active window contains a sequence file that has a `MainSequence` callback.

Execution window — A window in the sequence editor that displays the steps an execution runs. When execution is suspended, the execution window displays the next step to execute and provides single-stepping options. You also can view variables and properties in any active sequence context in the call stack.

expression — A formula that calculates a new value from the values of multiple variable or properties. In expressions, you can access all variables and properties in the sequence context that is active when TestStand evaluates the expression. The following is an example of an expression:

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

F

front-end callback — A common sequence that the sequence editor and run-time operator interfaces call. Front-end callbacks allow multiple applications to share the same implementation for a specific operation. TestStand installs the sequence file `FrontEndCallback.seq`, which contains the front-end callback sequence, `LoginLogout`.

front-end callback sequence file — A sequence file that contains front-end callbacks. TestStand installs the sequence file `FrontEndCallback.seq`, which contains the front-end callback sequence, `LoginLogout`.

front panel — The interactive user interface of a LabVIEW VI. Modeled from the front panel of physical instruments, it is composed of switches, slides, meters, graphs, charts, gauges, LEDs, and other controls and indicators.

G

global variable — TestStand defines two types of globals: sequence file globals and station globals. Sequence file globals are accessible by any sequence or step in the sequence file. Station globals are accessible by any sequence file loaded on the station. The values of station global variables are persistent across different executions and even across different invocations of TestStand.

GUI — *See* Run-Time Operator Interface.

H

hex — hexadecimal

highlight — The way in which input focus is displayed on a TestStand screen; to move the input focus onto an item.

I

in-process — When executable code runs in the same process space as the client, i.e. an ActiveX server in a dynamic-link library(DLL).

instance step type property — A built-in step property that exists in each step instance. Each step that you create with the step type has its own copy of the property. TestStand uses the value you specify for an instance property in the step type as the initial value of the property in each new step that you create. Normally, after you create a step, you can change the values of its instance properties.

interactive mode — When you run steps by selecting one or more steps in a sequence and choosing the **Run Selected Steps** or **Loop Selected Steps** items in the context menu or menu bar. The selected steps in the sequence execute, regardless of any branching logic that the sequence contains. The selected steps run in the order in which they appear in the sequence.

K

kill	To stop a running, terminating, or aborting execution by terminating the thread of the execution without any cleanup of memory. This can leave TestStand in an unreliable state.

L

LabVIEW	Laboratory Virtual Instrument Engineering Workbench. A program development application based on the programming language G and used commonly for test and measurement purposes
list box	A dialog box item that displays a list of possible choices.
local variable	A property of a sequence that holds a value or additional subproperties. Only a step within the sequence can directly access the property value.

M

main sequence	The sequence that initiates the tests on a UUT. The process model invokes the main sequence as part of the overall testing process. The process model defines what is constant about your testing process, whereas main sequences define the steps that are unique to the different types of tests you run.
MB	megabytes of memory
menu bar	Horizontal bar that contains names of main menus.
method	Performs an operation or function on an object.
MFC	Microsoft Foundation Class Library
model callback	A mechanism which allows a sequence file to customize the default behavior of a sequence in the process model.
model sequence file	A special type of sequence file that contains process model sequences. The sequences within the sequence file direct the high-level sequence flow of an execution when testing a UUT.

module adapter — A component that the TestStand engine uses to invoke code in another sequence or in a code module, such as LabVIEW. When invoking code in a code module, the adapter knows how to call it, and how to pass parameters to it.

N

named data type — A type of variable or property that you give a unique name. The data type usually contains multiple subproperties thus creating an arbitrarily complex data structure. All variables or properties that use the data type have the same data structure, but the values they contain can differ.

nested interactive execution — When you run steps interactively from an execution window for a normal execution that is suspended at a breakpoint. You can run steps only in the sequence and step group in which execution is suspended. The selected steps run within the context of the normal execution.

normal execution — When you start an execution in the sequence editor by selecting the **Run *Sequence Name*** item or one of the process model entry points from the Execute menu.

normal sequence file — Any sequence file containing sequences that test UUTs.

numeric property — A 64-bit floating-point value in the IEEE 754 format.

O

object — A service that an ActiveX server makes available to clients.

operator — A user profile that usually contains all privileges associated with operating a test station, but cannot debug sequence executions, edit sequence files, or configure user privileges, station options, report options, and database options.

out-of-process — When executable code does not run in the same process space as the client, such as an ActiveX server in an executable.

P

pop-up menus — *See* context menu

post actions — Actions that TestStand takes depending on the pass/fail status of the step or a custom condition the engine evaluates after executing a step. Post actions allow you to execute callbacks or jump to other steps after executing the step.

Post Step substep — A substep that the engine invokes after calling a step module. A Post Step substep might call a code module that compares the values the step module stored in step properties against limit values that the Edit substep stored in other step properties.

Pre Step substep — A substep that the engine invokes before calling the step module. For example, a Pre Step substep might call a code module that retrieves measurement configuration parameters and stores them into step properties for use by the step module.

preconditions — A set of conditions for a step that must be true for TestStand to execute the step during the normal flow of execution in a sequence.

process model — A series of operations before and after a test executive executes the sequence that performs the tests. Common operations include identifying the UUT, notifying the operator of pass/fail status, generating a test report, and logging results.

property — A container of information, which stores and maintains a setting or attribute of an object. A property can contain a single value, an array of values of the same type, or no value at all. A property can also contain any number of subproperties. Each property has a name.

property-array property — A property containing a value that is an array of subproperties of a single type. In addition to the array of subproperties, property-array properties can contain any number of subproperties of other types.

R

reference count — Each ActiveX object keeps track of the number of things that reference it. This allows the object to decide when to free the resources it uses.

reference property — *See* ActiveX reference property.

resource string — Text strings stored in an external file so you can alter the strings without directly altering the application.

root interactive execution	When you run selected steps from a Sequence File window in an independent execution. Root interactive executions do not invoke process models.
run mode	The mode in which you execute a step, such as normal, skip, force pass, force fail.
run-time error	An error condition that forces an execution to terminate. When the error occurs while running a sequence, TestStand jumps to the Cleanup step group, and the error propagates to any calling sequence up through to the top-level sequence.
run-time operator interface	A program that provides a graphical user interface for executing sequences at a production station. Sometimes the sequence editor and run-time operator interfaces are different aspects of the same program.
RTF	rich text format

S

s	seconds
sequence	A series of steps that you specify for execution in a particular order. Whether and when a step is executed can depend on the results of previous steps.
sequence context	A TestStand object that contains references to all global variables and all local variables and step properties in active sequences. The contents of the sequence context changes depending on the currently executing sequence and step.
sequence editor	A program that provides a graphical user interface for creating, editing, and debugging sequences.
sequence file	A file that contains the definition of one or more sequences.
single-valued property	A property that contains a single value. TestStand has four types of these properties: Number properties, String properties, Boolean properties, and ActiveX reference properties.
source code template	A set of source files that contain skeleton code, which serves as a starting point for the development of code modules for steps. TestStand uses the code template when the sequence developer clicks on the **Create Code** button on the Source Code tab in the Specify Module dialog box for a step.

standard named data type — A data type that TestStand defines and names. You can add subproperties to the standard data types, but you cannot delete any of their built-in subproperties. The standard named data types are `Path`, `Error`, and `CommonResults`.

station callback sequence file — A sequence file that contains the station callback sequences. Station callbacks run before and after the engine executes each step in any normal or interactive execution.

station globals — Variables that are persistent across different executions and even across different invocations of the sequence editor or run-time operator interfaces. The TestStand engine maintains the value of station global variables in a file on the run-time computer.

station model — A process model that you select to use for all sequence files for a station. The TestStand installation program establishes `TestStandModel.seq` as the default station model file. You can use the Station Options dialog box to select a different station model.

step — Any action, such calling a test module to perform a specific test, that you can include within a sequence of other actions.

step group — A set of steps in a sequence. A sequence contains the following groups of steps: Setup, Main, and Cleanup. When TestStand executes a sequence, the steps in the Setup group execute first, the steps in the Main group execute next, and the steps in the Cleanup group last.

step module — The code module that a step calls.

step property — A property of a step.

step result — A container property that contains a copy of the subproperties from the `Result` property of a step and additional execution information such as the name of the step and its position in the sequence. TestStand automatically creates a step result as each step executes and places the step result into a result list which TestStand uses to generate its reports.

step status — A string value that indicates the status of a step in an execution. Every step in TestStand has a `Result.Status` property. Although TestStand imposes no restrictions on the values to which the step or its code module can set the status property, TestStand and the built-in step types use and recognize a predefined set of values.

step type — A component that defines a set of custom step properties and standard behavior for each step of that type. All steps of the same type have the same properties, but the values of the properties can differ. Step types define their standard behaviors using substeps.

step-type-specific dialog box — A dialog box that step types display when their Edit substep is invoked. The dialog box lets you modify step properties that are specific to the step type. You invoke the dialog box with the menu item that appears in the context menu above **Specify Module**. For example, the Edit Limits item appears in the context menu for Numeric Limit test steps, and the Edit Pass/Fail Source item appears in the context menu for Pass/Fail test steps.

subsequence — A sequence that another sequence calls. You specify a subsequence call as a step in the calling sequence.

substep — Actions that a step type performs for a step besides calling the step module. You define a substep by selecting an adapter and specifying a module call. TestStand defines three different types of substeps: Edit substep, Pre Step substep, and Post Step substep.

substep module — The code module that a Edit, Pre Step, or Post Step substep calls.

T

technician — A user profile that usually contains all privileges associated with operating, and debugging sequences and sequences files, but cannot edit sequence files or configure user privileges, station options, report options, or database options.

template — *See* code template.

terminal — Object or region on a LabVIEW VI node through which data passes.

terminate — To stop an execution by halting the normal execution flow, and running all the Cleanup step groups in the sequences on the call stack.

test executive engine — A module or set of modules that provide an API for creating, editing, executing, and debugging sequences. A sequence editor or run-time execution operator interface uses the services of a test executive engine.

test module — A code module that performs a test.

U

Unit Under Test (UUT)	The device or component that you are testing.
user manager	The component of the TestStand engine that maintains a list of users, their login names and passwords, and their privileges. You can access the user manager from the User Manager window in the sequence editor.

V

variables	Properties that you can freely create in certain contexts. You can have variables that are global to a sequence file or local to a particular sequence. You can also have station global variables.
variables window	A window that shows the values of all the currently active variables or properties.
VI	Virtual instrument.
VI library	Special file of type `.LLB` that contains a collection of related VIs for a specific use.

W

watch window	A window that shows the values of user-selectable variables and expressions that are currently active.
window	A working area that supports specific tasks related to developing and executing programs.
wire	Tool used to define data paths between source and sink terminals.

Index

A

Abort (no cleanup) command, Debug menu, 4-18
Abort All (no cleanup) command, Debug menu, 4-18
Abort Immediately option, Run-Time Error dialog box, 6-26
aborting execution, 1-26
Action steps, 10-3 to 10-4
ActiveX Automation Adapter, 12-43 to 12-51
    configuring, 12-43 to 12-44
    running and debugging servers, 12-49
    specifying in Edit Automation Call dialog box, 12-44 to 12-48
        ActiveX Reference control, 12-45
        Automation Server control, 12-45
        Call Method or Access Property section, 12-46
        Create Object control, 12-45 to 12-46
        Object Class control, 12-45
        Parameters control, 12-46 to 12-47
    using with TestStand, 12-49 to 12-51
        compatibility issues with Visual Basic, 12-49 to 12-51
        registering servers, 12-49
    variant data types supported (table), 12-48
ActiveX Automation code modules, distributing, 16-12
Adapter Configuration dialog box, 12-2
adapters. *See* module adapters.
Adapters command, Configure menu, 3-2, 4-30
Add Watch command, Watch Expression pane, 6-13
All Sequences view context menu, 5-3 to 5-10
    Browse Sequence Context command, 5-3
    Insert Sequence command, 5-3
    Open Sequence command, 5-3
    Rename command, 5-3
    Sequence File Callbacks command, 5-9 to 5-10
    Sequence File Properties command, 5-6 to 5-9
    Sequence Properties command, 5-3 to 5-5
    View Contents command, 5-3
Allow Editing NI Installed Types option, Preferences tab, 4-26
Application Development Environment (ADE), 1-2
architecture of TestStand. *See* TestStand architecture overview.
Argument Expression control, Configure Call Executable dialog box, 10-19
arithmetic operators (table), 8-15
Array Bounds dialog box
    array sizing, 9-5
    empty arrays, 9-6
array function operators (table), 8-16
Array of submenu, 9-5
Array parameters, specifying for DLL Flexible Prototype Adapter, 12-9
array property, 1-10
arrays
    dynamic array sizing, 9-7
    empty arrays, 9-7
    modifying values, 9-11
    specifying array sizes, 9-5 to 9-6
assignment operators (table), 8-15
Attach to File control, General tab (Step Type Properties dialog box), 9-28
Auto-Load Library Configuration dialog box, 12-33
automatic result collection. *See* result collection.
Automatically Login Windows System User option, User Manager tab, 4-27 to 4-28

B

bitwise operators (table), 8-15
Break command, Debug menu, 4-18
Break All command, Debug menu, 4-18
Break on First Step command, Execute menu, 4-16
Break option
  Post Actions tab, 2-14, 5-23
  Run-Time Error dialog box, 6-27
Breakpoint option, Run Options tab, 5-21
breakpoints
  enabling/disabling, 4-20 to 4-21
  toggling, 5-16
Browse Sequence Context command
  All Sequences view context menu, 5-3
  Globals View context menu, 7-3
  Locals tab context menu, 5-32
  Parameters tab context menu, 5-29
  Sequence File Globals view context menu, 5-38
  Step Group context menu, 5-17
  View menu, 4-12 to 4-13
built-in properties
  definition, 1-11
  sequence properties, 1-16
  step properties, 1-11 to 1-12. *See also* Step Properties dialog box.
  step type properties, 9-24 to 9-25
built-in step types, 10-1 to 10-29. *See also* Step Type Properties dialog box; step types.
  any module adapter, 10-3 to 10-12
    Action steps, 10-3 to 10-4
    Numeric Limit Test, 10-6 to 10-9
    Pass/Fail Test, 10-4 to 10-6
    String Value Test, 10-9 to 10-12
  common custom properties, 10-1 to 10-2
  customizing, 10-2 to 10-3
  defining properties, 9-24 to 9-25. *See also* Step Type Properties dialog box.
  error occurred flag, 10-2
  module adapter not used, 10-14 to 10-29
    Call Executable, 10-18 to 10-21
    Goto, 10-28
    Label, 10-29
    Limit Loader, 10-21 to 10-28
    Message Popup, 10-15 to 10-18
    Statement, 10-14 to 10-15
  run-time errors, 10-2
  specific module adapters, 10-12 to 10-14
    Sequence Call, 10-12 to 10-14
  step status, 10-2
bulletin board support, A-1

C

Call Executable steps, 10-18 to 10-21
  Configure Call Executable dialog box, 10-19 to 10-20
  properties (figure), 10-20
  step properties defined, 10-20 to 10-21
Call sequence option, Post Actions tab, 2-14, 5-23
Call Stack pane, Execution window, 6-10 to 6-12
callback sequences, 1-23 to 1-24
  callback types (table), 1-23
  displaying
    in Callbacks dialog box, 5-9
    with Sequence File Callbacks command, 4-6 to 4-7
  engine callbacks, 1-23 to 1-24
  front-end callbacks
    customizing, 3-9 to 3-10
    overview, 1-24
  model callbacks
    customizing, 3-10
    defining, 1-18
    overview, 1-18 to 1-19
    purpose and use, 13-4
  restrictions on SequenceFileLoad and SequenceFileUnload callbacks, 5-10

Callbacks dialog box, 5-9 to 5-10
Cascade command, Window menu, 4-34
Category control, Module tab, 12-6 to 12-10
C/CVI Standard Prototype Adapter, 12-23 to 12-35
  Configuration dialog box (figure), 12-32
  configuring, 12-31 to 12-35
  example code module, 12-27 to 12-28
  executing code modules in external instance, 12-34 to 12-35
    debugging C source and DLL code modules, 12-35
  executing code modules in-process, 12-32 to 12-34
    debugging DLL code module, 12-33 to 12-34
    object and library code modules, 12-32 to 12-33
    source code modules, 12-33 to 12-34
  prototypes, 12-23 to 12-27
    step properties updated (table), 12-27
    tTestData structure member fields (table), 12-23 to 12-25
    tTestError structure member fields (table), 12-26
  specifying in Edit C/CVI Module Call dialog box, 12-28 to 12-31
    Module tab, 12-29 to 12-30
    Source Code tab, 12-30 to 12-31
Check Type option, Parameters tab context menu, 5-30
Check User Privileges option, User Manager tab, 4-27
Choose Code Template dialog box, 12-4
class step type properties, 9-24
Cleanup tab, step groups, 5-11
client sequence file, 1-18
Close command, File menu, 4-2
Close Completed Execution Displays command, Window menu, 4-34
Close Completed Execution Displays on Execution option, Preferences tab, 4-25
Close Tree View command, Step Group context menu, 5-17
code modules
  definition, 1-1
  executing in external instance, 12-34 to 12-35
    debugging C source and DLL code modules, 12-35
  executing in-process, 12-32 to 12-34
    debugging DLL code module, 12-33 to 12-34
    object and library code modules, 12-32 to 12-33
    source code modules, 12-33 to 12-34
code templates
  creating, 3-11, 9-35 to 9-36
  customizing, 9-35 to 9-36
  multiple templates per step type, 9-36
  source code templates
    for module adapters, 12-3 to 12-4
    for step types, 1-14
  template files for different adapters, 9-34 to 9-35
Code Templates tab, Step Type Properties dialog box, 9-34 to 9-40
  Add button, 9-38
  Create button, 9-37
  Create Code Templates dialog box, 9-38
  Edit button, 9-38
  Edit Code Template dialog box, 9-39 to 9-40
  illustration, 9-37
  Move Down button, 9-39
  Move Up button, 9-39
  overview, 9-34
  Remove button, 9-38
Comment control
  General tab, Step Properties dialog box, 5-18

General tab, Step Type Properties dialog box, 9-28
Sequence File Properties dialog box, 5-7
Sequence Properties dialog box, 5-5
CommonResults standard data type, 9-13
comparison operators (table), 8-15
Components directory, 3-4 to 3-6
customizing, 3-4 to 3-5
subdirectories (table), 3-5 to 3-6
configuration. *See also* customizing TestStand.
ActiveX Automation Adapter, 12-43 to 12-44
C/CVI Standard Prototype Adapter, 12-31 to 12-35
DLL Flexible Prototype Adapter, 12-5
LabVIEW Standard Prototype Adapter, 12-19
TestStand, 3-1 to 3-2
Configure menu, 3-1 to 3-2
sequence editor startup options, 3-1
configuration entry points, 13-4 to 13-5
Configure Call Executable dialog box, 10-19 to 10-20
Argument Expression control, 10-19
Executable Path control, 10-19
Exit Code Status Action control, 10-20
Initial Window State control, 10-20
Terminate Executable If Step Is Terminated Or Aborted control, 10-19
Time to Wait control, 10-19
Wait Condition control, 10-19
Configure menu, 3-1 to 3-2, 4-19 to 4-30
Adapters command, 3-2, 4-30
External Viewers command, 3-2, 4-30
Report Options command, 3-2, 4-30
Search Directories command, 3-2, 4-29 to 4-30
Station Options command, 3-2, 4-19 to 4-29
Configure Message Box Step dialog box, 10-16
configuring module adapters, 12-2 to 12-3
constants operators (table), 8-15
containers
container properties, 1-10
modifying values, 9-11
context menus
All Sequences view context menu, 5-3 to 5-10
Context tab context menu, 6-8 to 6-9
Globals View context menu, 7-2 to 7-4
Locals tab context menu, 5-31 to 5-32
Parameters tab context menu, 5-27 to 5-30
Profiles tab context menu, 11-6 to 11-7
purpose and use, 2-4
Sequence File Globals view context menu, 5-37 to 5-38
Step Group context menu, 5-14 to 5-27
Steps tab context menu, 6-6 to 6-7
User List context menu, 11-3 to 11-5
Context tab, Execution window, 6-7 to 6-9
Context tab context menu, 6-8 to 6-9
Properties command, 6-9
Refresh command, 6-9
View Contents command, 6-8
controlling sequences flow. *See* sequence flow, controlling.
copy, cut, and paste capabilities, sequence editor screen, 2-5
Copy command, Edit menu, 4-4
Create Code Templates dialog box, 9-38
Custom Condition Expression control, Post Actions tab, 5-23
custom data types, Properties dialog box for, 9-19 to 9-20
Apply Value to All Loaded Instances of the Type option, 9-19
Attach to File option, 9-20

changing single data type value, 9-9 to 9-10
illustration, 9-19
Value control, 9-19
Custom Data Types tab tree and list views, 9-13 to 9-17
creating and modifying custom data types, 9-16 to 9-17
list view, 9-15 to 9-16
tree view, 9-14 to 9-15
Value field, 9-16
custom named data types, 1-10
custom properties, 1-11
customer communication, *xxv*, A-1 to A-2
Customize command, Tools menu, 4-32 to 4-33
Customize Tool Menu dialog box, 4-32 to 4-33
customizing TestStand, 3-3 to 3-11
code templates, 3-11
creating string resource files, 3-6 to 3-8
data types, 3-8
directory structure, 3-3 to 3-6
engine and front-end callbacks, 3-9 to 3-10
process model, 3-10
process model callbacks, 3-10
run-time operator interfaces, 3-11
step types, 3-8
Tools menu, 3-9
users and user privileges, 3-11
cut and paste capabilities, sequence editor screen, 2-5
Cut command, Edit menu, 4-4

D

Data Source tab
Edit Numeric Limit Test dialog box, 10-8
Edit String Value Test dialog box, 10-11
data types, 9-3 to 9-20. *See also* types.
arrays
dynamic array sizing, 9-7
empty arrays, 9-7
specifying array sizes, 9-5 to 9-6
context menu items for using, 9-3 to 9-5
Insert Field, 9-4
Insert Global, 9-3, 9-4
Insert Local, 9-4
Insert Local submenu, 9-5
Insert Parameter, 9-3
Insert User, 9-4
submenus, 9-4 to 9-5
creating and modifying, 9-13 to 9-20
adding fields, 9-18
displaying and changing Value field, 9-16 to 9-17
Insert Custom Data Type submenu, 9-17 to 9-18
Insert Fields submenu, 9-18
new custom data type, 9-17 to 9-18
using Custom Data Types tab tree and list views, 9-13 to 9-17
using Properties dialog boxes, 9-19 to 9-20
custom named data types, 1-10
customizing, 3-8
displaying, 4-11, 9-8 to 9-9
modifying types and values, 9-9 to 9-11
arrays, 9-11
containers, 9-11
single values, 9-9 to 9-10
Numeric category data types (table), 12-7
Properties dialog box
custom data types, 9-19 to 9-20
data type fields, 9-20
standard named data types, 9-12 to 9-13
CommonResults, 9-13
Error, 9-13
Path, 9-12
purpose and use, 1-10

String category data types (table), 12-8
variant data types supported by ActiveX Automation Adapter (table), 12-48
Debug menu, 4-17 to 4-18
Abort (no cleanup) command, 4-18
Abort All (no cleanup) command, 4-18
Break command, 4-18
Break All command, 4-18
Resume command, 4-17
Resume All command, 4-18
Step Into command, 4-17
Step Out command, 4-17
Step Over command, 4-17
Terminate command, 4-18
Terminate All command, 4-18
debugging
ActiveX Automation servers, 12-49
C source and DLL code modules, 12-35
DLL code module, 12-34
DLLs, 12-12 to 12-13
LabVIEW Standard Prototype Adapter, 12-21 to 12-22
sequences execution, 2-17 to 2-18
Default Step Name Expression control, General tab (Step Type Properties dialog box), 9-27
Delete command, Edit menu, 4-4
Designate an Adapter control, General tab (Step Type Properties dialog box), 9-27
Designate an Icon control, General tab (Step Type Properties dialog box), 9-27
Destination control, Post Actions tab, 5-23
directory search paths, setting, 4-29 to 4-30
directory structure
process models, 13-1 to 13-2
TestStand, 3-3 to 3-6
Components directory, 3-4 to 3-6
NI and User subdirectories, 3-4
subdirectories (table), 3-3
Disable Properties tab, Step Type Properties dialog box, 9-32 to 9-33
illustration, 9-33
Precondition checkbox, 9-33
Specify Module checkbox, 9-33
Disable Result Recording for All Sequence option, Execution tab, 4-22
Disable Results for All Steps option, Sequence Properties dialog box, 5-4
Display Warning on Run Mode Changes in Execution Window option, Preferences tab, 4-25
distributing TestStand, 16-1 to 16-14
ActiveX Automation code modules, 16-12
creating installer for run-time engine, 16-1 to 16-5
DLL code modules, 16-8
installing customized engine, 16-6 to 16-8
LabVIEW, 16-6
LabWindows/CVI, 16-6 to 16-7
Visual Basic, 16-7 to 16-8
invoking custom engine installation, 16-5
LabVIEW run-time server, 16-12 to 16-14
distributing, 16-14
rebuilding, 16-13
LabVIEW test VIs, 16-9 to 16-12
packaging VIs and subVIs for sequence file, 16-10
saving VIs with full hierarchy, 16-11 to 16-12
saving VIs without full hierarchy, 16-10 to 16-11
object and static library code modules, 16-8 to 16-9
sequence files, 16-8
DLL code modules, distributing, 16-8
DLL Flexible Prototype Adapter, 12-4 to 12-13
configuring, 12-5
debugging DLLs, 12-12 to 12-13

Module tab of Edit DLL Call dialog box, 12-5 to 12-10
Array parameters, 12-9
Calling Convention control, 12-6
Category control, 12-6 to 12-10
DLL Pathname field, 12-6
Function control, 12-6
illustration, 12-5
Numeric parameters, 12-7
Object parameters, 12-9 to 12-10
Parameter section, 12-6
String parameters, 12-8
Source Code tab of Edit DLL Call dialog box, 12-10 to 12-12
adapter interpretation of ambiguous declarations (table), 12-12
Create Code button, 12-10
Edit Code button, 12-11
Pathname of Source file Containing Function control, 12-10
Verify Prototype button, 12-11
using MFC run-time library, 12-13
documentation
conventions used in manual, *xxiv-xxv*
organization of manual, *xxiii-xxiv*
related documentation, *xxv*
drag and drop capabilities, sequence editor screen, 2-5
dynamic array sizing, 9-7

E

Edit Automation Call dialog box, 12-44 to 12-48
ActiveX Reference control, 12-45
Automation Server control, 12-45
Call Method or Access Property section, 12-46
Create Object control, 12-45 to 12-46
illustration, 12-44
Object Class control, 12-45
Parameters control, 12-46 to 12-47
Edit C/CVI Module Call dialog box, 12-28 to 12-31
Module tab, 12-29 to 12-30
Extended Prototype, 12-29
Function Name, 12-29
illustration, 12-29
Module Pathname, 12-29
Module Type, 12-29
Pass Sequence Context, 12-30
Standard Prototype, 12-29
Source Code tab, 12-30 to 12-31
Create Code button, 12-31
Edit Code button, 12-31
illustration, 12-30
Pathname of Source File Containing Function control, 12-30
Edit Code command, Step Group context menu, 5-15
Edit Code Template dialog box, 9-39 to 9-40
Description control, 9-39
illustration, 9-39
Parameter Name/Value Mappings section, 9-40
Pass Sequence Context control, 9-39
Edit command
Step Group context menu, 5-15
User List context menu, 11-5
Edit DLL Call dialog box
Module tab, 12-5 to 12-10
Array parameters, 12-9
Calling Convention control, 12-6
Category control, 12-6 to 12-10
DLL Pathname field, 12-6
Function control, 12-6
illustration, 12-5
Numeric parameters, 12-7
Object parameters, 12-9 to 12-10
Parameter section, 12-6
String parameters, 12-8

Source Code tab, 12-10 to 12-12
adapter interpretation of ambiguous declarations (table), 12-12
Create Code button, 12-10
Edit Code button, 12-11
Pathname of Source file Containing Function control, 12-10
Verify Prototype button, 12-11
Edit Expression command, Watch Expression pane, 6-12
Edit LabVIEW VI Call dialog box, 12-20 to 12-21
Create Code button, 12-21
Edit Code button, 12-21
illustration, 12-20
Optional Parameters section, 12-20
VI Module Pathname control, 12-20
Edit Limit Loader Step dialog box
Layout tab, 10-23
Limits File tab, 10-22
Edit menu, 4-3 to 4-7
Copy command, 4-4
Cut command, 4-4
Delete command, 4-4
Paste command, 4-4
Select All command, 4-4
Sequence File Callbacks command, 4-6 to 4-7
Sequence File Properties command, 4-5 to 4-6
Sequence Properties command, 4-5
Edit Numeric Limit Test dialog box
Data Source tab, 10-8
Limits tab, 10-6
Edit Parameter Value dialog box, 12-47
Edit Pass/Fail Source dialog box, 10-5
Edit Paths dialog box (figure), 4-9
Edit Paths in Files dialog box (figure), 4-8
Edit Sequence Call dialog box
Edit Sequence Call tab, 12-37 to 12-39
illustration, 12-37
List Box in the Parameters Section, 12-38
Sequence control, 12-37 to 12-38
Specify Expressions for Pathname and Sequence, 12-37
Use Current File option, 12-38
Remote Execution tab, 12-39 to 12-41
illustration, 12-39
Remote Execution option, 12-40
Remote Host option, 12-40
Specify host by expression option, 12-40
Edit Statement Step dialog box, 10-15
Edit String Value Test dialog box
Data Source tab, 10-11
Limits tab, 10-10
Edit substep
overview, 1-13
Substeps tab, 9-30
Edit User dialog box, 11-5
Edit User Type command
Profiles tab context menu, 11-7
User List context menu, 11-5
electronic support services, A-1 to A-2
e-mail support, A-2
empty arrays, 9-7
Enable Breakpoints option, Execution tab, 4-20 to 4-21
Enable Tracing option, Execution tab, 4-21 to 4-22
engine callbacks, 6-20 to 6-22
available engine callbacks (table), 6-20 to 6-22
customizing, 3-9 to 3-10
definition, 6-20
examples of using, 6-22
overview, 1-23 to 1-24
entry points, 1-19 to 1-22
configuration entry points, 13-4 to 13-5
defining multiple entry points, 1-19
definition, 1-19

Execution Entry Point Sequence Model tab, 13-5 to 13-7
Entry Point Enabled Expression control, 13-6
Entry Point Ignores Client File control, 13-6
Entry Point Name Expression control, 13-6
Hide Entry Point Execution option, 13-7
illustration, 13-5
Load Stale Sequence Files Before Execution option, 13-7
Menu Hint control, 13-6
Save Modified Sequence Files Before Execution option, 13-7
Show Entry Point for All Windows option, 13-7
Show Entry Point Only in Editor option, 13-7
Show Entry Point When Client File Window is Active option, 13-7
Show Entry Point When Execution Window is Active option, 13-7
execution entry points
definition, 1-19
process models, 13-4
purpose and use, 6-2
flowchart of TestUUTs sequence in default process model (figure), 1-20
list of all sequences (figure), 1-22
set of steps for TestUUTs entry point (figure), 1-21
Error Out cluster, LabVIEW Standard Prototype Adapter, 12-16 to 12-17
element types and descriptions, 12-17
illustration, 12-16
Error standard data type, 9-13
escape codes for unprintable characters (table), 3-7
Executable Path control, Configure Call Executable dialog box, 10-19
Execute menu, 4-14 to 4-16
Break on First Step command, 4-16
Execution Entry Point List command, 4-14
Loop on Selected Steps command, 4-15 to 4-16
Restart command, 4-14
Run Active Sequence command, 4-14
Run Selected Steps command, 4-15
Tracing Enabled command, 4-16
execution, 1-24 to 1-26, 6-1 to 6-3. *See also* Execution window.
definition, 6-1
direct execution without process model, 6-2
engine callbacks, 6-20 to 6-22
interactive execution, 1-25, 6-3
normal execution, 1-25
overview, 1-24 to 1-25, 6-1
Preconditions dialog box, 5-32 to 5-35
run-time errors, 6-25 to 6-27
starting, 6-2 to 6-3
step execution (table), 6-23 to 6-24
step status property, 6-24
terminating and aborting executions, 1-26
Execution Entry Point dialog box, Model tab, 13-5 to 13-7
Entry Point Enabled Expression control, 13-6
Entry Point Ignores Client File control, 13-6
Entry Point Name Expression control, 13-6
Hide Entry Point Execution option, 13-7
illustration, 13-5
Load Stale Sequence Files Before Execution option, 13-7
Menu Hint control, 13-6
Save Modified Sequence Files Before Execution option, 13-7

Show Entry Point for All Windows option, 13-7
Show Entry Point Only in Editor option, 13-7
Show Entry Point When Client File Window is Active option, 13-7
Show Entry Point When Execution Window is Active option, 13-7
Execution Entry Point List command, 4-14
execution entry points
definition, 1-19
process models, 13-4
purpose and use, 6-2
execution pointer, 6-1
Execution tab, Station Options dialog box, 4-20 to 4-23
Disable Result Recording for All Sequence option, 4-22
Enable Breakpoints option, 4-20 to 4-21
Enable Tracing option, 4-21 to 4-22
Goto Cleanup On Sequence Failure option, 4-22
illustration, 4-20
Interactive Mode option, 4-22
Execution window, 6-3 to 6-14
areas in, 6-3
Call Stack pane, 6-10 to 6-12
Context tab, 6-7 to 6-9
context menu, 6-8 to 6-9
illustration, 6-8
definition, 1-25
example (figure), 2-7
overview, 2-6
Report tab, 6-9 to 6-10
result collection, 6-14 to 6-19
custom result properties, 6-16 to 6-17
loop results, 6-19
ResultList array (figure), 6-15
standard result properties, 6-17 to 6-18
subsequence results, 6-18 to 6-19
status bar, 6-12 to 6-13
Steps tab, 6-4 to 6-7
columns, 6-5 to 6-6
context menu, 6-6 to 6-7
debugging, 6-5
illustration, 6-4
tracing, 6-4 to 6-5
Threads selection ring, 6-3
Watch Expression pane, 6-12 to 6-13
Add Watch command, 6-13
Edit Expression command, 6-12
illustration, 6-12
Modify Value command, 6-13
Refresh command, 6-13
Exit Code Status Action control, Configure Call Executable dialog box, 10-20
Exit command, File menu, 4-3
exporting limit values. *See* Import/Export Sequence Limits dialog box.
Expression Browser dialog box, 8-13 to 8-14
illustration, 1-9, 8-13
Operators/Functions tab, 8-14
purpose and use, 8-13
expressions, 8-12 to 8-19
function operators (table), 8-16 to 8-18
levels of precedence (table), 8-19
operators (table), 8-15
purpose and use, 8-12
using values of variables and properties, 1-8 to 1-9
Expressions tab, Step Properties dialog box, 5-26 to 5-27
Post Expression control, 5-26
Pre Expression control, 5-26
Status Expression control, 5-26
External Viewers command, Configure menu, 3-2, 4-30

F

fax and telephone support numbers, A-2
Fax-on-Demand support, A-2
File menu, 4-1 to 4-3
    Close command, 4-2
    Exit command, 4-3
    Login command, 4-2
    Logout command, 4-2
    most recently opened files list, 4-3
    New command, 4-2
    Open command, 4-2
    Save command, 4-2
    Save All command, 4-3
    Save As command, 4-2
    Unload All Modules command, 4-3
Find Type command, View menu, 4-11
front-end callbacks
    customizing, 3-9 to 3-10
    overview, 1-24
FTP support, A-1
function operators for expressions (table), 8-16 to 8-18

G

General tab
    Sequence File Properties dialog box, 5-6 to 5-8
    Step Properties dialog box, 5-18 to 5-19
        Comment control, 5-18
        Edit button, 5-18
        illustration, 5-18
        Preconditions button, 5-19
        Specify Module button, 5-18 to 5-19
    Step Type Properties dialog box, 9-26 to 9-28
        Attach to File control, 9-28
        Comment control, 9-28
        Default Step Name Expression control, 9-27
        Designate an Adapter control, 9-27
        Designate an Icon control, 9-27
        illustration, 9-26
        Step Description Expression control, 9-27
global variables
    definition, 1-7
    lifetime and scope of sequence file global variables, 5-36
    station global variables
        persistence, 7-4 to 7-5
        special station globals, 7-5
Globals View context menu, 7-2 to 7-4
    Browse Sequence Context command, 7-3
    Go Up One Level command, 7-3
    Insert Global submenu, 7-2 to 7-3
    Properties command, 7-3
    Reload Station Globals command, 7-4
    Rename command, 7-3
    View Constants command, 7-3
Go Up One Level command
    Globals View context menu, 7-3
    Parameters tab context menu, 5-29
    Step Group context menu, 5-17
Go Up One level command
    Locals tab context menu, 5-32
    Sequence File Globals view context menu, 5-38
Goto built-in step type, 2-16
Goto Cleanup On Sequence Failure option
    Execution tab, 4-22
    Sequence Properties dialog box, 5-4
Goto destination option, Post Actions tab, 2-14, 5-23
Goto next step option, Post Actions tab, 2-14, 5-23
Goto steps, 10-28

H

Hide Entry Point Execution option, Execution Entry Point Sequence Model tab, 13-7
Hide User Manager Window option, User Manager tab, 4-27

I

Ignore option, Run-Time Error dialog box, 6-26
Ignore Run-time Errors option
  effect on execution, 6-25
  Run Options tab, Step Properties dialog box, 5-21
Ignore Termination option, Run Options tab, 5-22
Import/Export Limits command, Tools menu, 4-31, 10-26
Import/Export Sequence Limits dialog box, 10-26 to 10-28
  Append to End of file option, 10-28
  End of Data Marker control, 10-27
  Export button, 10-28
  First Row of Data Specifies Step Property for Each option, 10-27
  Format control, 10-27
  illustration, 10-26
  Import button, 10-27
  Sequence control, 10-26
  Sequence File indicator, 10-26
  Skip Rows That Begin With option, 10-27
  Source/Destination section, 10-27
  Start of Data Marker control, 10-27
Initial Window State control, Configure Call Executable dialog box, 10-20
Input buffer string control, LabVIEW Standard Prototype Adapter, 12-17
Insert Custom Data Type submenu, 9-17 to 9-18
Insert Field command, 9-4
Insert Fields submenu, 9-18
Insert Global submenu
  Globals View context menu, 7-2 to 7-3
  Sequence File Globals view context menu, 5-37
  using data types, 9-3
Insert Local command
  Locals tab context menu, 5-31
  using data types, 9-4
Insert Local submenu
  Array of submenu, 9-5
  using data types, 9-5
Insert New User dialog box, 11-4
Insert Parameter submenu
  Parameters tab context menu, 5-28
  using data types, 9-3, 9-4
Insert Profile command, Profiles tab context menu, 11-7
Insert Sequence command, All Sequences view context menu, 5-3
Insert Step submenu
  creating sequences, 2-10
  displaying and selecting step types, 9-21
  Step Group context menu, 5-14 to 5-15
Insert User command
  User List context menu, 11-4
  using data types, 9-3
Installation Wizard for TestStand Engine, 4-32, 16-1 to 16-4
instance step type properties, 9-25
interactive execution, 1-25, 6-3
Interactive Mode option, Execution tab, 4-22
Invocation Information cluster control, LabVIEW Standard Prototype Adapter, 12-17 to 12-18
Item Name Expression control, Menu tab, 9-29

K

keyboard actions for navigating lists and tree views (table), 2-3 to 2-4

L

Label step, 10-29
LabVIEW run-time operator interface, 15-4 to 15-5
  building standalone executable, 15-4 to 15-5
  distributing, 16-6
  top-level files (table), 15-4
LabVIEW run-time server, 16-12 to 16-14
  distributing, 16-14
  rebuilding, 16-13
LabVIEW Standard Prototype Adapter, 12-13 to 12-22
  configuring, 12-19
  debugging, 12-21 to 12-22
  specifying in Edit LabVIEW VI Call dialog box, 12-20 to 12-21
  structure, 12-13 to 12-18
    Error Out cluster, 12-16 to 12-17
    Input buffer string control, 12-17
    Invocation Information cluster control, 12-17 to 12-18
    Sequence Context control, 12-18
    Test Data cluster, 12-14 to 12-16
LabVIEW test VIs, distributing, 16-9 to 16-12
  packaging VIs and subVIs for sequence file, 16-10
  saving VIs with full hierarchy, 16-11 to 16-12
  saving VIs without full hierarchy, 16-10 to 16-11
LabWindows/CVI prototype adapter. *See* C/CVI Standard Prototype Adapter.
LabWindows/CVI run-time operator interface
  distributing, 16-6 to 16-7
  files in project file (table), 15-2 to 15-4
Language tab, Station Options dialog box, 4-28
Launch Report Viewer command, View menu, 4-13
Layout tab, Edit Limit Loader Step dialog box, 10-23
Limit Loader step, 10-21 to 10-28
  Edit Limit Loader Step dialog box
    Layout tab, 10-23
    Limits File tab, 10-22
  example sequence file (figure), 10-22
  Import/Export Sequence Limits dialog box, 10-26 to 10-28
  step properties (figure), 10-24
  step properties defined, 10-24 to 10-25
Limits File tab, Edit Limit Loader Step dialog box, 10-22
Limits tab
  Edit Numeric Limit Test dialog box, 10-6
  Edit String Value Test dialog box, 10-10
Load Option
  Run Options tab, Step Properties dialog box, 5-20
  Sequence File Properties dialog box, 5-7
local variables
  definition, 1-7
  lifetime of local variables, 1-15 to 1-16
  sequence local variables, 1-15
Locals tab (figure), 5-30
Locals tab context menu, 5-31 to 5-32
  Browse Sequence Context command, 5-32
  Go Up One level command, 5-32
  Insert Local command, 5-31
  Properties command, 5-32
  Rename command, 5-32
  View Contents command, 5-32
logical operators (table), 8-15
Login command, File menu, 4-2
Logout command, File menu, 4-2
Loop on Selected Steps command, Execute menu, 4-15 to 4-16
Loop on Selected Steps dialog box
  Loop Count tab, 4-15
  Stop Expression tab, 4-16

Loop Options tab, Step Properties dialog box, 5-24 to 5-25
illustration, 5-24
Loop Type control, 5-24 to 5-25
Record Result of Each Iteration option, 5-25
Loop Selected Steps command
Step Group context menu, 5-16
Steps tab context menu, 6-7
Loop Type control, Loop Options tab, 5-24 to 5-25

M

main sequence, 1-18
Main tab, step groups, 5-11
manual. *See* documentation.
menu bar, sequence editor. *See* sequence editor menu bar.
Menu Item Name Expression control, Substeps tab, 9-31 to 9-32
Menu tab, Step Type Properties dialog box, 9-28 to 9-29
illustration, 9-28
Item Name Expression control, 9-29
Singular Item Name Expression control, 9-29
Submenu Name Expression control, 9-29
Message Popup steps, 10-15 to 10-18
Configure Message Box Step dialog box, 10-16
properties (figure), 10-17
step properties defined, 10-17 to 10-18
MFC (Microsoft Foundation Class) run-time library, using with DLLs, 12-13
model callbacks
customizing, 3-10
defining, 1-18
overview, 1-18 to 1-19
purpose and use, 13-4
Model Option, Sequence File Properties dialog box, 5-8 to 5-9
Model tab
Sequence Properties dialog box, 13-5 to 13-7
Entry Point Enabled Expression control, 13-6
Entry Point Ignores Client File control, 13-6
Entry Point Name Expression control, 13-6
Hide Entry Point Execution option, 13-7
illustration, 13-5
Load Stale Sequence Files Before Execution option, 13-7
Menu Hint control, 13-6
Save Modified Sequence Files Before Execution option, 13-7
Show Entry Point for All Windows option, 13-7
Show Entry Point Only in Editor option, 13-7
Show Entry Point When Client File Window is Active option, 13-7
Show Entry Point When Execution Window is Active option, 13-7
Station Options dialog box
Allow Other Models option, 4-26
illustration, 4-26
Station Model field, 4-26
Use Station Model option, 4-26
Modify Numeric Value dialog box, 9-17
Modify Value command, Watch Expression pane, 6-13
module adapters, 12-1 to 12-51
ActiveX Automation Adapter, 12-43 to 12-51
available module adapters, 1-6, 12-2
C/CVI Standard Prototype Adapter, 12-23 to 12-35
configuring, 12-2 to 12-3

DLL Flexible Prototype Adapter, 12-4 to 12-13
LabVIEW Standard Prototype Adapter, 12-13 to 12-22
overview, 1-6 to 1-7, 12-1 to 12-2
Sequence Adapter, 12-35 to 12-43
source code templates, 12-3 to 12-4
Module tab
Edit C/CVI Module Call dialog box, 12-29 to 12-30
Extended Prototype, 12-29
Function Name, 12-29
illustration, 12-29
Module Pathname, 12-29
Module Type, 12-29
Pass Sequence Context, 12-30
Standard Prototype, 12-29
Edit DLL Call dialog box, 12-5 to 12-10
Array parameters, 12-9
Calling Convention control, 12-6
Category control, 12-6 to 12-10
DLL Pathname field, 12-6
Function control, 12-6
illustration, 12-5
Numeric parameters, 12-7
Object parameters, 12-9 to 12-10
Parameter section, 12-6
String parameters, 12-8
mouse and keyboard actions for navigating lists and tree views (table), 2-3 to 2-4

N

named data types, 1-10
nested interactive execution, 1-25, 6-3
New command, File menu, 4-2
NI subdirectory, 3-4
Numeric category data types (table), 12-7
numeric function operators (table), 8-16
Numeric Limit Test step, 10-6 to 10-9
comparison types (table), 10-7
Edit Numeric Limit Test dialog box
Data Source tab, 10-8
Limits tab, 10-6
properties (figure), 10-8
setting value of Step.Result.Numeric, 10-7 to 10-8
step properties defined, 10-9
Numeric parameters, specifying for DLL Flexible Prototype Adapter, 12-7

O

object and static library code modules, distributing, 16-8 to 16-9
Object parameters, specifying for DLL Flexible Prototype Adapter, 12-9 to 12-10
On Condition False control, Post Actions tab, 5-23
On Condition True control, Post Actions tab, 5-23
On Fail control, Post Actions tab, 5-23
On Pass control, Post Actions tab, 5-23
Open command, File menu, 4-2
Open Sequence command, All Sequences view context menu, 5-3
Open Tree View command, Step Group context menu, 5-16
operator interfaces. *See* run-time operator interfaces.
operators in expressions. *See* expressions.
Operators/Functions tab, Expression Browser dialog box, 8-14
Optimize Non-Reentrant Calls to this Sequence option, Sequence Properties dialog box, 5-5

P

Parameter section, Module tab, 12-6
Parameters tab (figure), 5-27

Parameters tab context menu, 5-27 to 5-30
Browse Sequence Context command, 5-29
Check Type option, 5-30
Go Up One Level command, 5-29
Insert Parameter submenu, 5-28
Pass By Reference command, 5-29
Properties command, 5-30
Rename command, 5-29
View Contents command, 5-28
Pass By Reference command, Parameters tab context menu, 5-29
Pass/Fail Test step, 10-4 to 10-6
Edit Pass/Fail Source dialog box, 10-5
properties (figure), 10-5
setting value of Step.Result.PassFail, 10-4
step properties defined, 10-5 to 10-6
paste capabilities, sequence editor screen, 2-5
Paste command, Edit menu, 4-4
Path standard data type, 9-12
Paths command, View menu, 4-8 to 4-10
Post Actions tab, Step Properties dialog box, 2-14, 5-22 to 5-23
Break option, 2-14, 5-23
Call sequence option, 2-14, 5-23
On Condition False control, 5-23
On Condition True control, 5-23
Custom Condition Expression control, 5-23
Destination control, 5-23
On Fail control, 5-23
Goto destination option, 2-14, 5-23
Goto next step option, 2-14, 5-23
illustration, 5-22
On Pass control, 5-23
Specify Custom Condition control, 5-23
Terminate execution option, 2-14, 5-23
Post Expression control, Expressions tab, 5-26
Post Step substep
definition, 1-13
Substeps tab, 9-30
Pre Expression control, Expressions tab, 5-26
Pre Step substep
definition, 1-13
Substeps tab, 9-30
Precondition option, Disable Properties tab, 9-33
Preconditions button
General tab, Step Properties dialog box, 5-19
Sequence Properties dialog box, 5-5, 5-33
Step Properties dialog box, 5-33
Preconditions dialog box, 2-14 to 2-15, 5-32 to 5-35
controlling sequence flow, 2-15
Copy button, 5-34
Cut button, 5-34
illustration, 2-15, 5-33
Insert AllOf button, 5-34
Insert AnyOf button, 5-34
Insert New Expression button, 5-34
Insert Step Status section
Insert Step Error, 5-35
Insert Step Executed, 5-35
Insert Step Fail, 5-35
Insert Step Pass, 5-35
list box items
AllOf block, 5-34
AnyOf block, 5-34
Arbitrary expression, 5-34
Step status expression, 5-34
Preferences tab, Station Options dialog box, 4-25 to 4-26
Allow Editing NI Installed Types, 4-26
Close Completed Execution Displays on Execution, 4-25
Display Warning on Run Mode Changes in Execution Window option, 4-25
illustration, 4-25
Prompt to Find Files option, 4-25
Save Before Running options, 4-26

Show Hidden Properties in Next Session option, 4-25
privileges for users, verifying, 11-11 to 11-12
process models, 1-17 to 1-22, 13-1 to 13-15. *See also* model callbacks.
client sequence file, 1-18
contents of default process model, 13-8 to 13-15
default sequences, 13-8 to 13-11
order of actions in Test UUTs entry point (table), 13-12 to 13-13
single pass entry point (table), 13-13
support files (table), 13-14 to 13-15
Test UUTs entry point (table), 13-12 to 13-13
customizing, 3-10
definition, 1-17
directory structure, 13-1 to 13-2
entry points, 1-19 to 1-22
flowchart of Test UUTs sequence in default process model (figure), 1-20
list of all sequences (figure), 1-22
set of steps for Test UUTs entry point (figure), 1-21
Execution Entry Point Sequence Model tab, 13-5 to 13-7
Entry Point Enabled Expression control, 13-6
Entry Point Ignores Client File control, 13-6
Entry Point Name Expression control, 13-6
Hide Entry Point Execution option, 13-7
illustration, 13-5
Load Stale Sequence Files Before Execution option, 13-7
Menu Hint control, 13-6
Save Modified Sequence Files Before Execution option, 13-7
Show Entry Point for All Windows option, 13-7
Show Entry Point Only in Editor option, 13-7
Show Entry Point When Client File Window is Active option, 13-7
Show Entry Point When Execution Window is Active option, 13-7
main sequence, 1-18
overview, 1-17
special editing capabilities for sequence files, 13-2 to 13-7
callback sequences, 13-4
entry point sequences, 13-4 to 13-7
marking sequence file in Sequence File Properties dialog box, 13-2
normal sequences, 13-3
station model, 1-17 to 1-18
Profiles tab, Users view, 11-5 to 11-7
Profiles tab context menu, 11-6 to 11-7
Edit User Type command, 11-7
Insert Profile command, 11-7
Prompt to Find Files option, Preferences tab, 4-25
properties. *See also* variables.
array property, 1-10
built-in properties
definition, 1-11
sequence properties, 1-16
step properties, 1-11 to 1-12
step type properties, 9-24 to 9-25
categories, 1-9 to 1-10
class step type properties, 9-24
container property, 1-10
custom properties, 1-11
custom step type properties, 9-23 to 9-24
definition, 1-7
displaying with Browse Sequence Context command, 4-12 to 4-13
instance step type properties, 9-25
property-array property, 1-10

single-valued property, 1-10
standard and custom named data types, 1-10
step properties, 1-7
using in expressions, 1-8 to 1-9
Properties command
Context tab context menu, 6-9
Globals View context menu, 7-3
Locals tab context menu, 5-32
Parameters tab context menu, 5-30
Sequence File Globals view context menu, 5-38
Step Group context menu, 2-11, 5-17
Steps tab context menu, 6-7
properties dialog boxes
custom data types, 9-19 to 9-20
data type fields, 9-20
Sequence File Properties dialog box, 5-6 to 5-9
Sequence Properties dialog box, 5-4 to 5-5
Step Properties dialog box, 2-12, 5-17 to 5-27
Step Type Properties dialog box, 9-25 to 9-40
property function operators (table), 8-16
property-array property, definition, 1-10
prototype adapters. *See* C/CVI Standard Prototype Adapter.

R

Record Result of Each Iteration option, Loop Options tab, 5-25
Record Results option, Run Options tab, 5-20
Refresh command
Context tab context menu, 6-9
Watch Expression pane, Execution window, 6-13
Reload Station Globals command, Globals View context menu, 7-4
Remote Execution tab. *See also* Sequence Adapter.
Edit Sequence Call dialog box, 12-39 to 12-41
illustration, 12-39
Remote Execution option, 12-40
Remote Host option, 12-40
Specify host by expression option, 12-40
Station Options dialog box, 4-28
Rename command
All Sequences view context menu, 5-3
Globals View context menu, 7-3
Locals tab context menu, 5-32
Parameters tab context menu, 5-29
Sequence File Globals view context menu, 5-38
Report Options command, Configure menu, 3-2, 4-30
Report Options dialog box, 14-4 to 14-10
Contents tab, 14-5 to 14-7
Append if File Already Exists option, 14-6
Disable Report Generation option, 14-5
illustration, 14-5
Include Execution Times option, 14-5 to 14-6
Include Output Values option, 14-6
Include Step Results option, 14-6
Include Test Limits option, 14-6
Report Colors control, 14-7
Report Format control, 14-6
Result Filtering Expression control, 14-6 to 14-7
Select a Report Generator for Producing the Report Body option, 14-7
overview, 14-4 to 14-5

Report File Pathname tab, 14-8 to 14-10
Add Time and Date to File Name option, 14-9
Base Name control, 14-9
Directory controls, 14-9
Force File Name to be Unique option, 14-9
Generate Report File Path button, 14-8
illustration, 14-8
New File for Each UUT option, 14-9
Prefix Sequence File Name to Report File Name option, 14-9
Specify Fixed Report File Path button, 14-10
Use Report Format Tag option, 14-10
Use Temporary File option, 14-8
Report tab, Execution window, 6-9 to 6-10
reports
ASCII format test report (figure), 14-4
generating test reports, 2-18 to 2-19
HTML test report (figure), 2-19, 14-3
implementation of test report capability, 14-1
Launch Report Viewer command, 4-13
using test reports, 14-2
resource string files. *See* string resource files.
Restart command, Execute menu, 4-14
result collection
automatic result collection, overview, 1-22
Execution window, 6-14 to 6-19
custom result properties, 6-16 to 6-17
loop results, 6-19
ResultList array (figure), 6-15
standard result properties, 6-17 to 6-18
subsequence results, 6-18 to 6-19
Resume command, Debug menu, 4-17
Resume All command, Debug menu, 4-18
root interactive execution, 1-25, 6-3
Run Active Sequence command, Execute menu, 4-14
Run Cleanup option, Run-Time Error dialog box, 6-26
Run Engine Installation Wizard, 16-1 to 16-4
Run Engine Installation Wizard command, Tools menu, 4-32
Run Mode ring, Run Options tab, 5-20
Run Mode submenu
Step Group context menu, 5-16
Force Fail, 5-16
Force Pass, 5-16
Normal, 5-16
Skip, 5-16
Steps tab context menu, 6-6
Run Options tab, Step Properties dialog box, 5-19 to 5-22
Breakpoint option, 5-21
Ignore Run-time Errors option, 5-21
Ignore Termination option, 5-22
illustration, 5-19
Load Option, 5-20
Record Results option, 5-20
Run Mode ring, 5-20
Sequence Call Trace Setting option, 5-21
Step Failure Causes Sequence Failure option, 5-21
Unload Option, 5-20
Run Selected Steps command
Execute menu, 4-15
Step Group context menu, 5-16
Steps tab context menu, 6-6
running sequences, 2-16 to 2-17
RunState subproperty, 8-4 to 8-7
RunState.InitialSelection subproperty, 8-10 to 8-11
RunState.Sequence subproperty and other Sequence objects, 8-9
RunState.Step subproperty and other Step objects, 8-10

run-time copy, created during execution, 6-1
Run-Time Error dialog box, 6-26 to 6-27
Abort Immediately option, 6-26
Break option, 6-27
Ignore option, 6-26
illustration, 6-26
Run Cleanup option, 6-26
Suppress this dialog for the remainder of this execution option, 6-27
run-time errors, 6-25 to 6-27
built-in step type, 10-2
description, 6-25
handling interactively, 6-26
Ignore Run-Time Errors option enabled, 6-25
overview, 2-16
run-time operator interfaces, 15-1 to 15-7
advantages, 2-20
compared with sequence editor, 6-1
considerations for customizing, 15-1 to 15-2
customizing, 3-11
definition, 1-2
distributing. *See* distributing TestStand.
LabVIEW interface, 15-4 to 15-5
LabWindows/CVI interface, 15-2 to 15-4
overview, 1-5
Visual Basic interface, 15-6 to 15-7

S

Save command, File menu, 4-2
Save All command, File menu, 4-3
Save As command, File menu, 4-2
Save Before Running options, Preferences tab, 4-26
Search Directories command, Configure menu, 3-2, 4-29 to 4-30
Select All command, Edit menu, 4-4
Sequence Adapter, 12-35 to 12-43
example parameters (table), 12-36
path resolution of sequence pathnames (table), 12-40
setting up TestStand as server for remote execution, 12-41 to 12-43
specifying in Edit Sequence Call dialog box, 12-36 to 12-41
Edit Sequence Call tab, 12-37 to 12-39
Remote Execution tab, 12-39 to 12-41
Sequence Call step, 10-12 to 10-14
Sequence Call Trace Setting option, Run Options tab, 5-21
sequence context, 8-1 to 8-11
definition, 1-7
first-level properties (table), 8-2
overview, 8-1
properties referring to objects that exist before and after current execution, 8-2
purpose and use, 8-11
subproperties, 8-3 to 8-11
RunState, 8-4 to 8-7
RunState.InitialSelection, 8-10 to 8-11
RunState.Sequence and other Sequence objects, 8-9
RunState.Step and other Step objects, 8-10
StationGlobals, 8-3
Sequence Context control, LabVIEW Standard Prototype Adapter, 12-18
sequence editor
compared with run-time operator interfaces, 6-1
configuring startup options, 3-1
context menus, 2-4
controlling sequence flow, 2-13 to 2-16
copy, cut, and paste capabilities, 2-5
creating sequences, 2-9 to 2-13
definition, 1-2
drag and drop capabilities, 2-5

Execution window. *See* Execution window.
lists and trees, 2-3 to 2-4
menu bar. *See* sequence editor menu bar.
mouse and keyboard actions for navigating lists and tree views (table), 2-3 to 2-4
overview, 1-5
screens, 2-1 to 2-5
Sequence File window, 2-6
Station Globals window, 2-8 to 2-9
status bar, 2-5
tabs, 2-3
toolbars, 2-5
Type Palette window, 2-7 to 2-8
Users window, 2-9
views, 2-2 to 2-3
windows, 2-2 to 2-5
sequence editor Execution window. *See* Execution window.
sequence editor menu bar, 4-1 to 4-34
Configure menu, 3-1 to 3-2, 4-19 to 4-30
Debug menu, 4-17 to 4-18
Edit menu, 4-3 to 4-7
Execute menu, 4-14 to 4-16
File menu, 4-1 to 4-3
overview, 2-5
Tools menu, 4-31 to 4-33
View menu, 4-7 to 4-13
Window menu, 4-34
sequence execution. *See* execution.
Sequence File Callbacks command
All Sequences view context menu, 5-9 to 5-10
Edit menu, 4-6 to 4-7
Sequence File Converters submenu, Tools menu, 4-31
Sequence File Documentation submenu, Tools menu, 4-31
Sequence File Globals view, 5-36 to 5-38
context menu, 5-37 to 5-38
illustration, 5-36
lifetime and scope of sequence file global variables, 5-36
Sequence File Globals view context menu, 5-37 to 5-38
Browse Sequence Context command, 5-38
Go Up One level command, 5-38
Insert Global submenu, 5-37
Properties command, 5-38
Rename command, 5-38
View Contents command, 5-38
Sequence File Properties command
All Sequences view context menu, 5-6 to 5-9
Edit menu, 4-5 to 4-6
Sequence File Properties dialog box, 5-6 to 5-9
Advanced tab, 5-8 to 5-9, 13-2
Comment control, 5-7
Full Path control, 5-6
General tab, 5-6 to 5-8
illustration, 5-6
Load Option, 5-7
Model Option, 5-8 to 5-9
Saved control, 5-6
Size control, 5-6
Type control, 5-8
Unload Option, 5-7
sequence file views, 5-1 to 5-39
All Sequences view, 5-2 to 5-10
All Sequences view context menu, 5-3 to 5-10
Browse Sequence Context command, 5-3
Insert Sequence command, 5-3
Open Sequence command, 5-3
Rename command, 5-3
Sequence File Callbacks command, 5-9 to 5-10

Sequence File Properties command, 5-6 to 5-9
Sequence Properties command, 5-3 to 5-5
View Contents command, 5-3
individual Sequence view, 5-10 to 5-32
Locals tab, 5-30
Locals tab context menu, 5-31 to 5-32
Main, Setup, and Cleanup tabs, 5-11 to 5-27
Parameters tab, 5-27
Parameters tab context menu, 5-27 to 5-30
Step Group context menu, 5-14 to 5-27
step group list view and tree view, 5-11
step group list view columns, 5-12 to 5-13
Sequence File Globals view, 5-36 to 5-38
context menu, 5-37 to 5-38
lifetime and scope of sequence file global variables, 5-36
Sequence File Types view, 5-39, 9-1
Sequence File window views, 5-1 to 5-2
Sequence File window
All Sequences view, 5-2 to 5-10
creating new sequence file, 2-9 to 2-10
example (figure), 2-6
individual Sequence view, 5-10 to 5-32
purpose and use, 2-6
Sequence File Globals view, 5-36 to 5-38
Sequence File Types view, 5-39
using View ring, 5-1
View ring contents (figure), 5-2
sequence files
client sequence file, 1-18
definition, 1-2
distributing, 16-8
overview, 1-16
storage of types in files, 1-17
types of files, 5-1
sequence flow, controlling, 2-13 to 2-16
Post Actions tab, 2-14
preconditions, 2-14 to 2-16, 5-32 to 5-35
status property values after execution completion (table), 2-13
sequence local variables, 1-7, 1-15
sequence parameters, 1-15. *See also* Parameters tab context menu.
Sequence Properties command
All Sequences view context menu, 5-3 to 5-5
Edit menu, 4-5
Step Group context menu, 5-17
Sequence Properties dialog box, 5-4 to 5-5
Comment control, 5-5
Disable Results for All Steps option, 5-4
Goto Cleanup on Sequence Failure control, 5-4
illustration, 5-4
Optimize Non-Reentrant Calls to this Sequence option, 5-5
Preconditions button, 5-5
sequence views. *See* sequence file views.
SequenceFileLoad callbacks, restrictions on, 5-10
SequenceFileUnload callbacks, restrictions on, 5-10
sequences, 1-15 to 1-16
built-in sequence properties, 1-16
callback sequences, 1-23 to 1-24
components, 1-15
creating, 2-9 to 2-13
debugging, 2-17 to 2-18
default sequences of process model, 13-8 to 13-11
definition, 1-2
lifetime of local variables, parameters, and custom step properties, 1-15 to 1-16

running, 2-16 to 2-17
sequence parameters, 1-15
step groups, 1-16
servers. *See* ActiveX Automation Adapter.
Set Next Step command, Steps tab context menu, 6-6
Setup tab, step groups, 5-11
Show Hidden Properties in Next Session option, Preferences tab, 4-25
Show Step in Context Tab command, Steps tab context menu, 6-7
single-valued property, 1-10
Singular Item Name Expression control, Menu tab, 9-29
software components of TestStand, 1-4 to 1-7
module adapters, 1-6 to 1-7
relationship between elements (figure), 1-4
run-time operator interfaces, 1-5
sequence editor, 1-5
test executive engine, 1-6
Source Code tab
Edit C/CVI Module Call dialog box, 12-30 to 12-31
Create Code button, 12-31
Edit Code button, 12-31
illustration, 12-30
Pathname of Source File Containing Function control, 12-30
Edit DLL Call dialog box, 12-10 to 12-12
adapter interpretation of ambiguous declarations (table), 12-12
Create Code button, 12-10
Edit Code button, 12-11
Pathname of Source file Containing Function control, 12-10
Verify Prototype button, 12-11
source code templates
for module adapters, 12-3 to 12-4
for step types, 1-14
special station global variables, 7-5
Specify Custom Condition control, Post Actions tab, 5-23
Specify Module button
General tab, Step Properties dialog box, 5-18 to 5-19
Substeps tab, Step Type Properties dialog box, 9-31
Specify Module command
dialog boxes for adapters (table), 9-22
Step Group context menu, 2-11, 5-15
Specify Module dialog box, Sequence Call step, 10-13
Specify Module option, Disable Properties tab, 9-33
standard named data types, 9-12 to 9-13
CommonResults, 9-13
Error, 9-13
Path, 9-12
purpose and use, 1-10
Statement steps, 10-14 to 10-15
station global variables
persistence, 7-4 to 7-5
special station globals, 7-5
Station Globals command, View menu, 4-8
Station Globals Types view, 9-1
Station Globals window, 7-1 to 7-4
Globals View context menu, 7-2 to 7-4
illustration, 7-1
overview, 2-8 to 2-9
View ring, 7-2
station model, 1-17 to 1-18
Station Options command, Configure menu, 4-19
Station Options dialog box, 4-19 to 4-29
Execution tab, 4-20 to 4-23
Disable Result Recording for All Sequence option, 4-22
Enable Breakpoints option, 4-20 to 4-21
Enable Tracing option, 4-21 to 4-22

Goto Cleanup On Sequence Failure option, 4-22
illustration, 4-20
Interactive Mode option, 4-22
Language tab, 4-28
Model tab
Allow Other Models option, 4-26
illustration, 4-26
Station Model field, 4-26
Use Station Model option, 4-26
overview, 3-2
Preferences tab, 4-25 to 4-26
Allow Editing NI Installed Types option, 4-26
Close Completed Execution Displays on Execution option, 4-25
Display Warning on Run Mode Changes in Execution Window option, 4-25
illustration, 4-25
Prompt to Find Files option, 4-25
Save Before Running options, 4-26
Show Hidden Properties in Next Session option, 4-25
Remote Execution tab, 4-28
Time Limits tab, 4-23 to 4-24
illustration, 4-23
Time Limits Setting ring, 4-23
When Time Expires options, 4-24
User Manager tab, 4-27 to 4-28
Automatically Login Windows System User option, 4-27 to 4-28
Check User Privileges option, 4-27
Hide User Manager Window option, 4-27
illustration, 4-27
User Manager File display, 4-27
StationGlobals subproperty, 8-3
status bar
Execution window, 6-12 to 6-13
sequence editor screen, 2-5
Status Bar command, View menu, 4-13
Status Expression control, Expressions tab, 5-26
status property of steps, 6-24
Step Description Expression control, General tab (Step Type Properties dialog box), 9-27
step execution (table), 6-23 to 6-24
Step Failure Causes Sequence Failure option, Run Options tab, 5-21
Step Group context menu, 5-14 to 5-27
Browse Sequence Context command, 5-17
Close Tree View command, 5-17
Edit Code command, 5-15
Edit command, 5-15
Go Up One Level command, 5-17
Insert Step submenu, 5-14 to 5-15
Loop Selected Steps command, 5-16
Open Tree View command, 5-16
Properties command, 5-17
Run Mode submenu, 5-16
Run Selected Steps command, 5-16
Sequence Properties command, 5-17
Specify Module command, 5-15
Step Properties dialog box, 5-17 to 5-27
Toggle breakpoint command, 5-16
View Contents command, 5-17
step groups
list view and tree view, 5-11
list view columns, 5-12 to 5-13
Main, Setup, and Cleanup tabs, 5-11 to 5-27
overview, 1-16
Step Into command, Debug menu, 4-17
step module, 1-1
Step Out command, Debug menu, 4-17
Step Over command, Debug menu, 4-17
Step Properties dialog box, 5-17 to 5-27
Expressions tab, 5-26 to 5-27
Post Expression control, 5-26
Pre Expression control, 5-26

Status Expression control, 5-26
General tab, 5-18 to 5-19
Comment control, 5-18
Edit button, 5-18
illustration, 5-18
Preconditions button, 5-19
Specify Module button, 5-18 to 5-19
illustration, 2-12
Loop Options tab, 5-24 to 5-25
illustration, 5-24
Loop Type control, 5-24 to 5-25
Record Result of Each Iteration option, 5-25
overview, 2-12
Post Actions tab, 2-14, 5-22 to 5-23
Break option, 2-14, 5-23
Call sequence option, 2-14, 5-23
On Condition False control, 5-23
On Condition True control, 5-23
Custom Condition Expression control, 5-23
Destination control, 5-23
On Fail control, 5-23
Goto destination option, 2-14, 5-23
Goto next step option, 2-14, 5-23
illustration, 5-22
On Pass control, 5-23
Specify Custom Condition control, 5-23
Terminate execution option, 2-14, 5-23
Run Options tab, 5-19 to 5-22
Breakpoint option, 5-21
Ignore Run-time Errors option, 5-21
Ignore Termination option, 5-22
illustration, 5-19
Load Option, 5-20
Record Results option, 5-20
Run Mode ring, 5-20
Sequence Call Trace Setting option, 5-21
Step Failure Causes Sequence Failure option, 5-21
Unload Option, 5-20
step status property, 6-24
Step Type Properties dialog box, 9-25 to 9-40
Code Templates tab, 9-34 to 9-40
Add button, 9-38
Create button, 9-37
Create Code Templates dialog box, 9-38
creating and customizing template files, 9-35 to 9-36
Edit button, 9-38
Edit Code Template dialog box, 9-39 to 9-40
illustration, 9-37
Move Down button, 9-39
Move Up button, 9-39
multiple templates per step type, 9-36
overview, 9-34
Remove button, 9-38
template files for different adapters, 9-34 to 9-35
Disable Properties tab, 9-32 to 9-33
illustration, 9-33
Precondition checkbox, 9-33
Specify Module checkbox, 9-33
General tab, 9-26 to 9-28
Attach to File control, 9-28
Comment control, 9-28
Default Step Name Expression control, 9-27
Designate an Adapter control, 9-27
Designate an Icon control, 9-27
illustration, 9-26
Step Description Expression control, 9-27
Menu tab, 9-28 to 9-29
illustration, 9-28
Item Name Expression control, 9-29

Singular Item Name Expression control, 9-29
Submenu Name Expression control, 9-29
overview, 9-25
Substeps tab, 9-30 to 9-32
Create button, 9-31
Delete button, 9-31
Description string indicator, 9-31
Edit substep, 9-30
illustration, 9-31
Menu Item Name Expression control, 9-31 to 9-32
Post Step substep, 9-30
Pre Step substep, 9-30
Specify Module button, 9-31
View Contents button, 9-41
step types, 1-12 to 1-14, 9-21 to 9-41. *See also* built-in step types; types.
creating and modifying custom step types, 9-22 to 9-41
built-in step type properties, 9-24 to 9-40
copying and renaming built-in step types, 9-22
custom step type properties, 9-23 to 9-24
displaying built-in step types in Type Palette window (figure), 9-23
displaying custom properties with View Contents button, 9-41
overview, 3-8
definition, 1-12
displaying with Find Type command, 4-11
Insert Step submenu, 9-21
overview, 1-12 to 1-13
predefined step types, 1-14
source code templates, 1-14
storing in Type Palette window, 9-41
substeps, 1-13
using, 9-21 to 9-22
steps
built-in step properties, 1-11 to 1-12
definition, 1-1
overview, 1-11
properties, 1-7
Steps tab, Execution window, 6-4 to 6-7
columns, 6-5 to 6-6
debugging, 6-5
illustration, 6-4
tracing, 6-4 to 6-5
Steps tab context menu, 6-6 to 6-7
Loop Selected Steps command, 6-7
Properties command, 6-7
Run Mode submenu, 6-6
Run Selected Steps command, 6-6
Set Next Step command, 6-6
Show Step in Context Tab command, 6-7
Toggle Breakpoint command, 6-6
String category data types (table), 12-8
string function operators (table), 8-16 to 8-17
String parameters, specifying for DLL Flexible Prototype Adapter, 12-8
string resource files, 3-6 to 3-8
default resource string files, 3-6
escape codes (table), 3-7
format, 3-7 to 3-8
search order for directories, 3-6
String Value Test step, 10-9 to 10-12
Edit String Value Test dialog box
Data Source tab, 10-11
Limits tab, 10-10
properties (figure), 10-11
setting value of Step.Result.String, 10-10 to 10-11
step properties defined, 10-12
subdirectories for TestStand (table), 3-3
Submenu Name Expression control, Menu tab, 9-29

subproperties of sequence context, 8-3 to 8-11
RunState, 8-4 to 8-7
RunState.InitialSelection, 8-10 to 8-11
RunState.Sequence and other Sequence objects, 8-9
RunState.Step and other Step objects, 8-10
StationGlobals, 8-3
subsequence, 1-2
substeps, 1-13
Substeps tab, Step Type Properties dialog box, 9-30 to 9-32
Create button, 9-31
Delete button, 9-31
Description string indicator, 9-31
Edit substep, 9-30
illustration, 9-31
Menu Item Name Expression control, 9-31 to 9-32
Post Step substep, 9-30
Pre Step substep, 9-30
Specify Module button, 9-31

T

technical support, A-1 to A-2
telephone and fax support numbers, A-2
templates. *See* code templates; Code Templates tab.
Terminate command, Debug menu, 4-18
Terminate All command, Debug menu, 4-18
Terminate Executable If Step Is Terminated Or Aborted control, Configure Call Executable dialog box, 10-19
Terminate execution option, Post Actions tab, 2-14, 5-23
terminating executions, 1-26
Test Data cluster, LabVIEW Standard Prototype Adapter, 12-14 to 12-16
element types and descriptions (table), 12-15
illustration, 12-14
older elements (table), 12-16
test executive engine, 1-2, 1-6
test module, 1-1
test reports. *See* reports.
TestStand
configuring, 3-1 to 3-2
customizing, 3-3 to 3-11
directory structure, 3-3 to 3-6
TestStand architecture overview, 1-1 to 1-26
building blocks, 1-7 to 1-26
automatic result collection, 1-22
callback sequences, 1-23 to 1-24
process models, 1-17 to 1-22
sequence executions, 1-24 to 1-26
sequence files, 1-16 to 1-17
sequences, 1-15 to 1-16
steps, 1-11 to 1-14
variables and properties, 1-7 to 1-11
capabilities and concepts, 1-2 to 1-3
general test executive concepts, 1-1 to 1-2
software components, 1-4 to 1-7
module adapters, 1-6 to 1-7
relationship between elements (figure), 1-4
run-time operator interfaces, 1-5
sequence editor, 1-5
test executive engine, 1-6
Threads selection ring, Execution window, 6-3
Tile command, Window menu, 4-34
time function operators (table), 8-17
Time Limits tab, Station Options dialog box, 4-23 to 4-24
illustration, 4-23
Time Limits Setting ring, 4-23
When Time Expires options, 4-24
Time to Wait control, Configure Call Executable dialog box, 10-19
Toggle Breakpoint command
Step Group context menu, 5-16
Steps tab context menu, 6-6

toolbars, sequence editor screen, 2-5
Toolbars command, View menu, 4-13
Tools menu, 4-31 to 4-33
    Customize command, 4-32 to 4-33
    customizing, 3-9
    Import/Export Limits command, 4-31
    Run Engine Installation Wizard command, 4-32
    Sequence File Converters submenu, 4-31
    Sequence File Documentation submenu, 4-31
    Update Automation Identifiers command, 4-31 to 4-32
tracing, enabling/disabling, 4-21 to 4-22
Tracing Enabled command, Execute menu, 4-16
TS.CurrentUser station global variable, 7-5
TS.LastUserName station global variable, 7-5
Type Conflict in File dialog box, 9-3
Type control, Sequence File Properties dialog box, 5-8
Type Palette command, View menu, 4-8
Type Palette window
    displaying built-in step types (figure), 9-23
    illustration, 2-8
    overview, 2-7
    purpose and use, 9-2
    storing custom step types, 9-41
types. *See also* data types; step types.
    storage in files and memory, 9-2
    windows and views that display types
        Sequence File Types view, 9-1
        Station Globals Types view, 9-1
        Type Palette window, 9-2
        User Types view, 9-1 to 9-2
Types view. *See* User Manager window.

U

unit under test (UUT), 1-2
Unload All Modules command, File menu, 4-3
Unload Option
    Run Options tab, Step Properties dialog box, 5-20
    Sequence File Properties dialog box, 5-7
unprintable characters, escape codes for (table), 3-7
Update Automation Identifiers command, Tools menu, 4-31 to 4-32
User List context menu, 11-3 to 11-5
    Edit command, 11-5
    Edit User Type command, 11-5
    Insert User command, 11-4
User Manager command, View menu, 4-8
User Manager tab, Station Options dialog box, 4-27 to 4-28
    Automatically Login Windows System User option, 4-27 to 4-28
    Check User Privileges option, 4-27
    Hide User Manager Window option, 4-27
    illustration, 4-27
    User Manager File display, 4-27
User Manager Types view, 11-7 to 11-11
    adding new properties and privileges, 11-10 to 11-12
    illustration, 11-7
    overview, 9-1 to 9-2
    Standard Data Types tab, 11-8 to 11-9
User Manager Users view, 11-2 to 11-7
    illustration, 11-2
    Profiles tab, 11-5 to 11-7
        illustration, 11-6
        Profiles tab context menu, 11-6 to 11-7
    User List tab, 11-3 to 11-5
        Edit User dialog box, 11-5
        illustration, 11-3
        Insert New User dialog box, 11-4
        User List context menu, 11-3 to 11-5

User Manager window, 11-1 to 11-12
overview, 11-1
sequence editor Users window, 2-9
verifying user privileges, 11-11 to 11-12
any user, 11-12
current user, 11-11 to 11-12
User subdirectory, 3-4

V

Value field
custom data types, 9-16
Modify Numeric Value dialog box, 9-17
variables. *See also* properties.
definition, 1-7
displaying with Browse Sequence Context command, 4-12 to 4-13
global
definition, 1-7
lifetime and scope of sequence file global variables, 5-36
local
definition, 1-7
lifetime of local variables, 1-15
sequence local variables, 1-15
sequence context of, 1-7
standard and custom named data types, 1-10
station global variables
persistence, 7-4 to 7-5
special station globals, 7-5
using in expressions, 1-8 to 1-9
View Constants command, Globals View context menu, 7-3
View Contents button, Step Type Properties dialog box, 9-41
View Contents command
All Sequences view context menu, 5-3
Context tab context menu, 6-8
Locals tab context menu, 5-32
Parameters tab context menu, 5-28
Sequence File Globals view context menu, 5-38
Step Group context menu, 5-17
View menu, 4-7 to 4-13
Browse Sequence Context command, 4-12 to 4-13
Find Type command, 4-11
Launch Report Viewer command, 4-13
Paths command, 4-8 to 4-10
Station Globals command, 4-8
Status Bar command, 4-13
Toolbars command, 4-13
Type Palette command, 4-8
User Manager command, 4-8
views. *See also* sequence file views.
sequence editor screen, 2-2 to 2-3
Visual Basic compatibility issues, 12-49 to 12-51
Visual Basic run-time operator interface
distributing, 16-7 to 16-8
top-level files (table), 15-6 to 15-7

W

Wait Condition control, Configure Call Executable dialog box, 10-19
Watch Expression pane, Execution window, 6-12 to 6-13
Add Watch command, 6-13
Edit Expression command, 6-12
illustration, 6-12
Modify Value command, 6-13
Refresh command, 6-13
Window menu
Cascade command, 4-34
Close Completed Execution Displays command, 4-34
open windows list, 4-34
Tile command, 4-34