
TestStand User Manual



Mess- und Prüftechnik. Die Experten.

**Ihr Ansprechpartner /
Your Partner:**

dataTec AG
E-Mail: info@datatec.eu
>>> www.datatec.eu



Authorized
Distributor

2025-07-23



Contents

TestStand User Manual	34
TestStand Overview	35
TestStand Key Features.....	35
Components of a TestStand System	35
Installing TestStand	37
Activating and Licensing TestStand	39
New Features and Changes	44
Updates and Changes for TestStand Extended Support Versions	54
TestStand 2022 Q4.....	54
TestStand 2021 SP1.....	57
TestStand 2021.....	59
TestStand 2020.....	60
TestStand 2019.....	61
TestStand 2017 SP1.....	68
TestStand 2017.....	69
Year-Based and Major.Minor Version Equivalents	80
Updates to Default Database Schemas	81
Migrating to the Current Version of TestStand	83
Migrating TestStand 2010 SP1 or Earlier Result Processing Options to the Current Version of TestStand	84
Manually Migrating Result Processing Options from Any Version of TestStand.....	86
Fundamentals	88
Getting Started with TestStand.....	88
Introduction to TestStand	89
Loading and Running Sequences.....	96
Editing Steps in a Sequence.....	105
Debugging Sequences.....	112
Using Variables and Properties.....	116
Using Callback Sequences	122
Adding Users and Setting Privileges	127
Interactive Executions.....	130
Calling Sequences Dynamically	132

Customizing Reports	136
Calling LabVIEW VIs	141
Calling LabWindows/CVI Code Modules	151
Parts of a TestStand Application	159
General Test Executive Concepts	162
TestStand Building Blocks	163
Sequences	163
Sequence Files	164
Step Groups	165
Sequence Local Variables	166
Sequence File Global Variables	167
Sequence Parameters	167
Steps	168
Step Types	168
Built-In Step Properties	169
Custom Step Properties	170
Code Modules	170
Communicating with Hardware Drivers	171
Properties	172
Built-In and Custom Properties	172
PropertyObject Attributes	173
Variables	176
Station Global Variables	176
Expressions	178
Expression Operators	179
Levels of Precedence in Operators	182
Special Constant Values	183
Special String Characters	185
Process Models	187
Specifying Process Model Files	187
Main Sequence and Client Sequence File	188
Entry Points	188
Standard and Custom Data Types	189
Automatic Result Collection	189
Callback Sequences	190
Workspaces	191

Selecting a Source Code Control Provider	193
TestStand File Extensions	195
Executions	196
Sequence Context	198
Accessing the Sequence Context	199
Top-Level Properties	200
Station Globals TS Subproperty	201
RunState Subproperties	202
SequenceFile Object Subproperties	207
Sequence Object Subproperties	208
RunState.Step and Other Step Objects	209
InitialSelection Subproperties	210
Executing Sequences	211
Using Execution Entry Points	212
Executing a Sequence Directly	213
Interactively Executing Steps	213
Debugging Executions	214
Terminating and Aborting Executions	215
Checking for Suspended or Stopped Execution within Code Modules ...	216
Executing Sequences Remotely	218
Configuring Remote Executions	219
Configuring TestStand as a Server for Remote Executions	220
Setting Windows System Security on a Remote Server	221
DCOM Settings for Remote Servers	221
Windows Firewall Settings for Remote Servers	225
Settings Windows System Security on a Local Client	226
DCOM Settings for a Local Client	226
Windows Firewall Settings for the Local Client	227
Step Execution	229
Result Collection	232
Standard Result Properties	234
Custom Result Properties	235
Exceptions to Custom Result Properties	237
Subsequence Results	238
Loop Results	239
Engine Callbacks	240

Step Status	240
Run-Time Errors	243
Using TestStand on SMP Systems	244
Built-In Step Types	247
Creating Steps from Step Types	248
Step Types You Can Use with Any Module Adapter	248
Step Types That Work with Specific Module Adapters	249
Step Types That Do Not Use Module Adapters	250
Using the Property Loader Step Type	250
Importing and Exporting with a Legacy Source	250
Importing Data to Non-Existing Properties	251
Validate and Verify Runtime Behavior of Import	252
Using Alias Names in Property Loader	253
Property Loader Plugins - Database Table Schemas	254
Property Loader Plugins - File Formats	258
Using the Synchronization Step Type	265
Setting Up TestStand for Accessing Synchronization Objects Remotely ..	265
DCOM Settings for the Remote Computer - Setting Up TestStand for	
Accessing Synchronization Objects Remotely	265
DCOM Settings for the Local Computer - Setting Up TestStand for	
Accessing Synchronization Objects Remotely	268
Windows Firewall Settings for the Remote Computer - Setting Up	
TestStand for Accessing Synchronization Objects Remotely	269
Windows 10/8.1/7 Firewall Settings - Windows Firewall Settings	
for the Remote Computer - Setting Up TestStand for Accessing	
Synchronization Objects Remotely	269
Windows Vista Firewall Settings - Windows Firewall Settings for	
the Remote Computer - Setting Up TestStand for Accessing	
Synchronization Objects Remotely	271
Windows Firewall Settings for the Local Computer - Setting Up	
TestStand for Accessing Synchronization Objects Remotely	271
Windows 10/8.1/7 Firewall Settings - Windows Firewall Settings	
for the Local Computer - Setting Up TestStand for Accessing	
Synchronization Objects Remotely	272
Windows Vista Firewall Settings - Windows Firewall Settings for	
the Local Computer - Setting Up TestStand for Accessing	
Synchronization Objects Remotely	273

Synchronized Sections	274
Mismatched Sections - Synchronized Sections	275
Nested Sections - Synchronized Sections	275
Using the Data Streams Step Type	276
Attaching a CSV Output Stream to the Current Execution	276
Reading from CSV Files with Data Streams Step Types	277
Writing to CSV Files with Data Streams Step Types	278
Module Adapters	278
Search Directories	279
Configuring Search Directories for Deployment	281
Calling DLLs from TestStand	281
Loading Subordinate DLLs	282
Choosing Between the Shared or Side-by-Side Version of the LabWindows/ CVI Run-Time Engine	283
Using Side-by-Side Versions of the LabWindows/CVI Run-Time Engine with TestStand	284
Reading Parameter Information for a DLL	285
Debugging DLLs	286
Adding Type Libraries to LabWindows/CVI DLLs	288
Generating Type Library Information to Include in a DLL	288
Specifying String Buffer Size in a Type Library	289
Module Adapter Parameter Mapping Guidelines	290
Passing Existing TestStand Container Variables to LabVIEW	291
Updating Cluster Element Mapping	293
LabVIEW Adapter	294
Caveats for Configuring the LabVIEW Adapter to Use a LabVIEW Run-Time Engine or Other Executable Server	295
Normalization of Numeric Values Passed to Enumeration Parameter Values	297
Enumeration Parameter Value Update Behavior	298
LabVIEW Data Types in TestStand	299
Legacy Configurations for LabVIEW Code Modules in TestStand	304
Error Reporting for Broken VIs in the LabVIEW Run-Time Engine	305
Version Independent Run-Time Engine Support	306
Switching between VIs and Packed Project Libraries	308
Updating VI Dependencies	309

Calling LabVIEW VIs That Invoke .NET Code in TestStand	310
Executing Source-Only VIs in LabVIEW Runtime Engine	311
Build Source Files and Execute	313
LabWindows/CVI Adapter	318
LabWindows/CVI Step Execution Locations	319
Out-of-Process LabWindows/CVI Code Module Execution	320
In-Process LabWindows/CVI Code Module Execution	320
LabWindows/CVI Data Type Equivalencies in TestStand	322
Mapping LabWindows/CVI Parameters	323
Calling Legacy LabWindows/CVI Code Modules	324
Updating Step Properties	325
Example Legacy LabWindows/CVI Code Module	326
C/C++ DLL Adapter	328
Exporting Class Methods and Functions in Microsoft Visual Studio	329
Parsing Parameters from Source Code	330
.NET Adapter	331
Managing Versions of .NET to Use with TestStand	332
Viewing XML Help for .NET Code Modules	333
Debugging .NET Assemblies	334
Enabling Code Module Hot Reloading During Step-Into Debugging in TestStand	335
Options for Stepping Out of a Microsoft Visual Studio Assembly ...	336
Managed Code Considerations for TestStand and .NET 8	336
ActiveX/COM Adapter	339
HTBasic Adapter	339
Sequence Adapter	340
Python Adapter	342
Python Adapter Known Limitations	343
Loading and Unloading Python Modules	344
Handling Python Code Execution when Importing Modules	344
Python Interpreter Sessions	345
Console for Interpreter Sessions in Python	346
Python Namespaces	346
Debugging Python Modules	347
Updating Python Modules while Debugging	349
Database Logging	350

TestStand Database Fundamentals	351
Databases and Tables	351
Database Sessions	352
Microsoft ADO, OLE DB, and ODBC Database Technologies	353
Structured Query Language (SQL)	354
Format Strings for Database Date Values	355
Data Links	358
Example Data Link and Result Table Setup for Microsoft Access	359
Database Options—Specifying a Data Link and Schema	360
Database Viewer—Creating Result Tables	361
Configuring Logging to Custom Database	362
Connecting to a Database with the ODBC Administrator	363
ODBC Data Source Administrator Dialog Box	363
TestStand Database Result Tables	364
Default TestStand Table Schemas	365
Using Legacy Database Schemas	366
UUT_RESULT Table Schema	366
STEP_RESULT Table Schema	367
STEP_SEQCALL Table Schema	369
PROP_RESULT Table Schema	370
PROP_BINARY Table Schema	371
PROP_ANALOGWAVEFORM Table Schema	371
PROP_DIGITALWAVEFORM Table Schema	372
PROP_NUMERICLIMIT Table Schema	373
Adding Support for Other Database Management Systems	374
Customizing Database Logging Expressions	376
On-the-Fly Database Logging	377
Logging Binary Strings in Databases	378
Database Known Issues	379
Logging Enumerations as Strings	384
Logging Attribute Values	386
Generating and Customizing TestStand Reports	387
Choosing the Appropriate Report Generation Strategy	387
Maximizing Test System Throughput	389
Minimizing Report File Size	391
Including All Necessary Report Content	392

Interoperating with Other Processes and Systems.	394
Supporting Post-Failure Information Recovery.	396
Generating the Report for the Current UUT before Testing the Next UUT.	397
Generating Multiple Reports for Each UUT.	398
Generating and Viewing the Report during Test Sequence Execution.	400
Use Case: Generate Different Types of Reports from the Same Data Set. .	401
Use Case: Log Data at One Location but Generate the Report at a Separate Location.	403
Generating a Test Report.	404
Customizing Reports with Expressions.	405
Specify Report File Path by Expression.	408
Customizing the Graph Control in Reports.	415
Failure Chain in Reports.	419
Batch Reports.	419
Property Flags That Affect Reports.	419
Storing Additional Information for UUTs and Test Stations in Process Models. .	420
Escaping Special Characters in Reports.	423
On-the-Fly Report Generation.	423
Including Binary Strings in Reports.	425
Retrieving Binary String Values from Reports and Databases.	426
Displaying Measurement Data as Graphs.	426
Displaying Measurement Data in an XML Report.	428
Including Hyperlinks in Reports.	429
Comparing Report Format Features.	430
Handling Large Amounts of Data.	437
Generating PDF Reports.	437
XML Reports.	438
Transforming XML Reports.	439
Transforming XML Content to HTML Output.	439
Distributing XML Reports.	441
XML Report Style Sheets.	441
Sections in XML Report Style Sheets.	443
Customizing XML Report Style Sheets.	450
Adding Columns and Rows to an XML Report.	451
Adding Images and Text to an XML Report Header or Footer. .	454

Adding Sequence Call Step Results to an XML Report without Indentation	457
Changing the Display Format of Execution Time in an XML Report	460
Customizing the UUT Report Header in an XML Report	462
Excluding Results from an XML Report Based on Step Properties. .	465
Modifying Cell Background Color Based on Step Status in an XML Report	468
Rendering HTML Elements and New Line Characters in an XML Report	472
ATML Test Results Reports	473
ATML	475
ATML Test Results 6.01 and 5.0 Report Structure	476
ATML Test Results 2.02 Report Structure	478
ATML Report Style Sheets	480
Customizing ATML Test Results 6.01 or 5.0 Report Style Sheets	481
Adding Columns and Rows to an ATML Test Results 6.01 or 5.0 Report	482
Adding Images and Text to an ATML Test Results 6.01 or 5.0 Report Header or Footer	485
Adding Step Results to an ATML Test Results 6.01 or 5.0 Report without Indentation	488
Adding Sequence Call Step Results to an ATML Test Results 6.01 or 5.0 Report without Indentation	490
Changing the Display Format of Execution Time in an ATML Test Results 6.01 or 5.0 Report	492
Customizing the UUT Report Header for an ATML Test Results 6.01 or 5.0 Report	494
Modifying Cell Background Color Based on Step Status in an ATML Test Results 6.01 or 5.0 Report	494
Rendering HTML Elements and New Line Characters in an ATML Test Results 6.01 or 5.0 Report	497
Customizing the ATML Test Results 2.02 Report Style Sheet	498
Adding Columns to an ATML Test Results 2.02 Report	499
Enabling or Disabling Table Indentation in an ATML Test Results 2.02 Report	500

Displaying or Hiding TestStand-Specific Information in an ATML Test Results 2.02 Report	500
Displaying the Fully Expanded ATML Test Results 2.02 Report in Plain HTML.....	503
Customizing the UUT Report Header for an ATML Test Results 2.02 Report	503
Customizing the ATML Test Results Report Generator.....	504
Using NI DataFinder with TestStand	505
Managing Users	506
Privileges	507
Accessing Privilege Settings for the Current User.....	508
Accessing Privilege Settings for Any User	508
Defining Custom Privileges	509
Process Model Architecture	509
TestStand Process Models	511
Features Common to All TestStand Process Models.....	513
Configuration Entry Points.....	513
Engine Callback	514
Plug-in Utility Sequences.....	514
Sequential Process Model	515
Execution Entry Points in the Sequential Process Model	516
Test UUTs Execution Entry Point in the Sequential Process Model..	516
Single Pass Execution Entry Point in the Sequential Process Model	517
Model Callbacks in the Sequential Process Model	518
Parallel Process Model	523
Main Execution Entry Points in the Parallel Process Model	524
Test UUTs Execution Entry Point in the Parallel Process Model.....	524
Single Pass Execution Entry Point in the Parallel Process Model....	525
Hidden Execution Entry Points in the Parallel Process Model	526
Test UUTs – Test Socket Entry Point Execution Entry Point in the Parallel Process Model	527
Single Pass - Test Socket Execution Entry Point in the Parallel Process Model	528
Utility Sequences in the Parallel Process Model	529
Model Callbacks in the Parallel Process Model	531

Batch Process Model	536
Main Execution Entry Points in the Batch Process Model	537
Test UUTs Execution Entry Point in the Batch Process Model	538
Single Pass Execution Entry Point in the Batch Process Model	541
Hidden Execution Entry Points in the Batch Process Model.....	543
Test UUTs – Test Socket Entry Point Execution Entry Point in the Batch Process Model.....	544
Single Pass - Test Socket Entry Point in the Batch Process Model...	546
Utility Sequences in the Batch Process Model.....	548
Model Callbacks in the Batch Process Model.....	550
Batch Controller and Test Socket Synchronization Architecture.....	556
Batch Controller and Test Socket Synchronization Restrictions	558
Implementing Batch Controller and Test Socket Synchronization for a Custom Model Plug-in.....	558
Report Generation Functions and Sequences.....	561
Default Report Generator Process Model Plug-in Sequences that Generate Reports.....	562
Default Process Model Sequences and Functions for Generating Step Results	563
Model Callbacks You Can Override to Alter Report Generation	564
Manually Selecting Process Models.....	565
Process Model Support Files	565
Process Model Plug-In Architecture.....	569
Configuring Process Model Plug-ins	573
Programmatically Updating a Configuration File	574
Model Plug-in Categories	575
Automatically Adding or Removing a Plug-in File to or from Configurations	575
Creating Process Model Plug-ins.....	576
Process Model Types	580
Process Model Thread Types.....	580
Step Results.....	582
Structure of Plug-in Sequence Files.....	584
Model Plug-in Entry Points.....	584
Model Plugin – Configure Standard Options	590
Model Plugin – Configure Additional Options	591

Model Plugin – Initialize	592
Model Plugin – Begin	593
Model Plugin – Pre Batch	594
Model Plugin – Pre UUT	595
Model Plugin – Batch Start	596
Model Plugin – UUT Start	597
Model Plugin – OnTheFly Step Results	598
Model Plugin – UUT Done	600
Model Plugin – Batch Done	601
Model Plugin – Post UUT	602
Model Plugin – Post Batch	603
Model Plugin – End	604
Order of Entry Point Execution at Run Time	605
Plug-in Entry Point Parameters	606
Avoid Batch Synchronization Sections in Process Model Plug-in Entry Points	614
Model Plug in Client-File Callbacks	614
Invoking Model Callbacks	615
Enabling or Disabling Editing of NI_ModelPlugin Properties	617
Integrating Model Plug-ins with a Custom Process Model	617
Debugging Result Processing Plug-ins with Offline Results	619
Migrating Process Model Customizations to TestStand 2012 or Later	619
Migrating Custom Results Processing Components	621
Migrating Changes to Settings of the Default Result Processors ...	622
Migrating Modifications to Process Model Callbacks	622
Migrating Additional Results from the Client Sequence	624
Migrating Modifications to Built-in Result Processors	625
Migrating Custom Result Processors	627
Migrating Other Custom Built-in Process Model Components	630
Migrating Model Options Files	630
Migrating Customizations to Process Model Callbacks	631
Migrating Customizations to Process Model Dialog Boxes	632
Migrating Error Handling Customizations	633
Migrating Custom Process Model Components	634
64-bit TestStand and Migrating from 32-bit TestStand	634

Choosing between 32-bit TestStand and 64-bit TestStand.....	635
Activating 32-bit TestStand and 64-bit TestStand.....	635
Features 64-Bit TestStand does not Support.....	636
Driver Support for 64-bit TestStand.....	636
64-bit Architecture Offers More Memory.....	637
Bitness-Conditional Code for 32-bit TestStand or 64-bit TestStand.....	638
Representing Pointers in Sequences in 32-bit TestStand and 64-bit TestStand.....	638
Representing Pointer-Sized Numbers in Sequences in 32-bit TestStand and 64-bit TestStand.....	639
Using an Object Reference to Represent Pointer-Sized Numbers in Sequences in 32-bit TestStand and 64-bit TestStand.....	640
Using a TestStand Number with Representation Based on Bitness to Represent Pointer-Sized Numbers in Sequences in 32-bit TestStand and 64-bit TestStand	640
Using Two Separate Variables to Represent Pointer-Sized Numbers in Sequences in 32-bit TestStand and 64-bit TestStand.....	641
Locating the Correct Code Module in 32-bit TestStand and 64-bit TestStand ..	641
Using TestStand Search Directories to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand	641
Using the \$(Platform) Path Macro to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand	643
Adapter and Code Module Support for 64-bit TestStand	644
Sequence Call Adapter Support for 64-bit TestStand.....	646
CPU Affinity Mask Support for 64-bit TestStand	646
Remote Execution in 32-bit TestStand and 64-bit TestStand	647
.NET Code Module Support for 64-bit TestStand	648
AnyCPU Assemblies in 32-bit TestStand and 64-bit TestStand.....	649
Platform-Specific Assemblies in 32-bit TestStand and 64-bit TestStand	649
ActiveX/COM Code Module Support for 64-bit TestStand	651
In-Process COM Servers Support in 32-bit TestStand and 64-bit TestStand	651
Out-of-Process COM Servers in 32-bit TestStand and 64-bit TestStand ..	651
C/C++ DLL and LabWindows/CVI DLL Support for 64-bit TestStand.....	652

Locating the Correct DLL in 32-bit TestStand and 64-bit TestStand	653
Bitness-Conditional Code for DLLs with Different 32-bit Interfaces and 64-bit Interfaces	654
Pointers and Handles in 32-bit TestStand and 64-bit TestStand	655
LabVIEW Code Module Support for 64-bit TestStand	656
Migrating LabVIEW Code Modules from 32-bit TestStand to 64-bit TestStand	657
Simultaneously Supporting 32-bit TestStand and 64-bit TestStand	657
Express VIs and Property Node Calls in 32-bit TestStand and 64-bit TestStand	657
Cross-Bitness Support using the LabVIEW Development System	658
Call Executable Step Type Support for 64-bit TestStand	659
CPU Affinity Step Type Support for 64-bit TestStand	659
Synchronization Object Support for 64-bit TestStand	660
Session Manager Support for 64-bit TestStand	660
Python Code Module Support for 64-bit TestStand	661
Deploying 32-bit TestStand and 64-bit TestStand Systems	661
Installer Issues for 32-bit TestStand and 64-bit TestStand	662
API Updates for 64-bit TestStand	663
TestStand Environments	664
Defining an Environment	665
Environment Behavior	667
Choosing an Environment	668
General Environment Limitations	670
Changing Environments	671
Enabling Large Address Aware Functionality in 32-bit TestStand	671
Extending TestStand	673
Customizing the Tools Menu	674
Configuring Sequence Editor and User Interface Startup Options	674
TestStand Directory Structure	677
TestStand Directory	678
Components Directory	679
TestStand Public Directory	681
Copying Read-Only Files to Modify	682

<TestStand Application Data> Directory	683
TestStand Local Application Data Directory	686
Type Concepts	686
Storing Types in Files and Memory	687
Password-Protecting Types	687
Modifying Types	689
Modifying Types Without Altering the Version Number	690
Type Versioning	692
Managing Type Revisions	693
Avoiding Unwanted Type Version Propagation	694
Using Type Palette Files to Restrict Automatic Type Conflict Resolution ..	695
Type Palette Files	696
Data Types	698
Creating Instances of Data Types	698
Specifying Array Sizes	700
Dynamic Array Sizing	700
Specifying an Empty Array	701
Modifying Type Instances, Values, and Type Definitions	701
Assigning Values to Object References	703
Using Standard Named Data Types	704
Error and CommonResults	704
Path	706
Expression	707
NI_TDMSReference	707
Creating Custom Data Types	708
Property Settings Specific to Data Types	709
Custom Step Types	709
Creating Custom Step Types	709
Properties Common to All Step Types	710
Custom Properties of Step Types	711
Configuring the Adapter Icon Color for a Custom Step Type	711
Using Substeps	712
Multiple Code Templates per Step Type	714
Creating and Customizing Code Template Files	714
Template Files for Different Development Environments	715
Managing Step Type Version Compatibility	717

Adding Step Types to Surround Selection With Context Menu	718
Customizing Process Models and Callbacks	719
Modifying Process Model Sequence Files	720
Normal Sequences	721
Using Callback Sequences to Modify Process Models	721
Entry Point Sequences	722
Modifying Callbacks	724
Modifying Engine Callbacks	724
List of Engine Callbacks	725
Caveats for Using Engine Callbacks	729
Modifying Front-End Callbacks	730
Creating Custom User Interfaces	731
TestStand UI Controls	732
Using TestStand UI Controls in Different Environments	733
Using TestStand UI Controls in LabVIEW	733
Configuring the TestStand UI Controls for a LabVIEW User Interface	735
Special Considerations for a LabVIEW Custom User Interface	736
Using TestStand UI Controls in LabWindows/CVI	737
Binding a TestStand Custom User Interface to a Side-by-Side Version of the LabWindows/CVI RTE	739
Configuring the TestStand UI Controls for a LabWindows/CVI User Interface	740
Handling Variants in LabWindows/CVI User Interfaces	740
Using TestStand UI Controls in Microsoft Visual Studio	741
Using TestStand UI Controls in Microsoft Visual C++	743
Getting Started with User Interface Development	744
Manager Controls	745
Application Manager	745
SequenceFileView Manager	746
ExecutionView Manager	746
Visible Controls	747
Connecting Manager Controls to Visible Controls	749
View Connections	749
List Connections	750

Command Connections	751
Information Source Connections	753
Caption Connections	753
Image Connections	754
Numeric Value Connections	755
Specifying and Changing Control Connections	755
Handling Events	756
Events Typical Applications Handle	757
Handling Events in a LabVIEW User Interface	759
Handling Events in a LabWindows/CVI User Interface	761
Startup and Shutdown	761
Starting and Shutting Down TestStand from a LabVIEW User Interface ..	762
Starting and Shutting Down TestStand from a LabWindows/CVI User	
Interface	763
Starting and Shutting Down TestStand from a .NET User Interface ..	764
License Checking	768
Editor Versus Operator Interface Applications	769
Creating Editor Applications	769
User Interface Development Best Practices	770
Localizing User Interfaces	770
Documenting Custom User Interfaces	771
User Interface Application Styles	772
Single Window Applications	772
Multiple Window Applications	773
No Visible Window Applications	775
Persisting Application Settings	775
Specifying the Configuration File Location	776
Adding Custom Application Settings	777
Deploying a User Interface	777
Advanced User Interface Development	778
TestStand Utility Functions Library	778
Adding Assembly References in Microsoft Visual Studio	779
Managing Menus and Menu Items	780
Managing Menu Bars and Menu Event Handling in a LabVIEW User	
Interface	782

Managing Menu Bars in a LabWindows/CVI User Interface.....	784
Implementing Command-Line Arguments for a User Interface.....	785
Authenticode Signatures for Microsoft Windows	785
Application Manifests	787
Making Dialog Boxes Modal to TestStand.....	787
Migrating User Interfaces to .NET Core	788
Example User Interfaces.....	790
Required Visual Studio Components for Building Custom C# User Interfaces	792
Caveats for Using the LabVIEW Example User Interfaces	792
Sequence File Translators	794
Creating a Sequence File Translator DLL	796
Example Sequence File Translators	797
Versioning Translators and Custom Sequence Files.....	798
Deploying Translators and Custom Sequence Files	799
Creating String Resource Files.....	800
String Resource File Format.....	801
Using Enumerations in TestStand	802
Getting Started Creating a TestStand Enumeration	802
Creating an Enumeration Data Type and Defining the Enumerators	803
Range Checking: Strict and Loose Enumerations.....	803
Flags Enumerations and Bitwise OR Semantics	804
Creating an Enumeration Instance.....	804
TestStand Enumerations Reference Guide.....	804
Enumerations as Numbers.....	804
Enumerations as PropertyObjects	805
Enumerations in TestStand Expressions.....	808
Using Enumerations with the TestStand Adapters	809
Enumeration Options	813
Enumerations and Type Merging	813
Invalid TestStand Enumerations.....	814
Developing TestStand Systems.....	815
Crash Recovery for TestStand Applications	815
TestStand Crash Log and Dump File	816
Registering a Crash Recovery Sequence.....	818
TestStand Tools and Utilities	819

TestStand Sequence Editor	819
Customizing TestStand Sequence Editor Toolbars and Menus	820
Configuring Keyboard Shortcuts in the TestStand Sequence Editor	821
Locking Toolbars, Menus, and Panes	822
TestStand Sequence Editor Keyboard Navigation	822
Drag and Drop Step Creation in the TestStand Sequence Editor	824
Creating a Subsequence from Selected Steps - Sequence File Window	826
Step Type Combination	828
Importing Specifications into TestStand Using the Specifications Pane	829
Importing Specifications from SLE Specification Manager	829
Importing Specifications from Specification Files	831
Updating Values from an SLE Server in TestStand	833
SLE Attributes, Conditions, and Specifications	834
Using Imported Specifications in TestStand	838
Exporting Values to an SLE Server from TestStand	840
TestStand Deployment Utility	841
TestStand File Diff and Merge Utility	843
TestStand Offline Results Processing Utility	846
Logging Offline Results Processing Utility Messages to a File	849
XML Packaging Utility	850
TestStand Sequence Analyzer	853
TestStand Sequence Analyzer Application	854
Analyzing Files	857
Analyzing a Sequence File or Workspace File in the TestStand Sequence Editor	858
Analyzing Multiple TestStand Files Using the TestStand Sequence Analyzer ..	860
TestStand Sequence Analyzer Rules Descriptions	863
Viewing Examples in the Sequence Analyzer	869
Creating Custom Rules in the TestStand Sequence Analyzer	870
Editing Custom Rules and Analysis Modules	873
Localizing Custom Rules in the TestStand Sequence Analyzer	873
Creating Analysis Modules for Custom Rules	874
Prototypes for Analysis Modules	874
Implementing Analysis Modules	876
Making a Custom Rule Configurable in the TestStand Sequence Analyzer	879

Adding Rule Settings for Custom Rules in the TestStand Sequence Analyzer . .	880
Accessing Rule Settings in Analysis Modules	880
Creating Rule Configuration Modules in the TestStand Sequence Analyzer . . .	882
Accessing Rule Configuration Data in Rule Configuration Modules and Analysis Modules	884
Debugging Analysis Modules and Rule Configuration Modules.	885
Calling Rule Configuration Modules for Custom Rules	886
Deploying Custom Rules and Analysis Modules	887
Sequence Analyzer Install Utility	889
Converting LabVIEW Analysis Modules and Rule Configuration Modules to DLLs. . .	890
Execution Profiler.	891
Types of Information the Profiler Records	891
Item Names	894
Resource Names - Resource Usage Profiler	895
Determining Resource Usage	896
Auto Scheduling and Thread Synchronization Operations.	897
Displaying Additional Timing Information	898
Programmatically Displaying and Configuring the Profiler	899
Excluding Items from Profiling	900
Profiler Window Graphs	900
Item Usage per Thread Graph	901
Thread Usage per Item Graph	902
Managing Profiler Overhead	902
Database Viewer Application.	903
Executing SQL Commands in the SQL Editor Window.	904
Customizing the Database Viewer User Interface	907
Printing and Exporting Data in Database Viewer	908
Configuring Connection Strings	908
Database Viewer Known Issues.	912
Issues with Using the MSDASQL Provider in Database Viewer.	912
Instrument Drivers.	914
IVI Drivers	914
Plug and Play Instrument Drivers.	915
Route Specification Strings	916
Sequence File Converter	918

TestStand Version Selector	921
Using the TestStand Version Selector	924
Session Manager	925
Instrument Sessions	926
Why Use Instrument Sessions?	927
Instrument Session Types	928
Instrument Session Names	929
Instrument Session Categories	930
Instrument Session Name Configuration	931
VXIplug&play Base Interface	934
Limitations with Sharing Instrument Sessions Between Processes	935
All Development Environments	935
Using Instrument Sessions in a Development Environment	936
LabWindows/CVI	937
LabVIEW	937
Microsoft Visual C++ and Other C++ Environments	937
TestStand	938
Initialize with Options	938
Session Locking	940
Attaching Additional Data to a Session	940
What Software Do I Install to Use Instrument Sessions?	941
TestStand Migration Utility	941
Process Model Changes in TestStand 2012 or Later and the Migration Utility ..	944
Nigel, AI Advisor	946
Deploying TestStand Systems	949
Before You Begin	949
Deployment Process Overview	953
Identifying Components to Deploy	955
Choosing a Deployment Mechanism	961
Choosing a TestStand Deployment Utility Deployment Build Output	966
Deploying with Environments Using the TestStand Deployment Utility ..	970
Building an Installer with the TestStand Deployment Utility	972
Uninstalling a TestStand Deployment	974
Building a Package Distribution with the TestStand Deployment Utility .	975
Configuring the Package Settings.	976
Building a Package Distribution with NI Package Builder	977

Building an Installer with Third-Party Tools	978
Network-Based Deployment Mechanisms	980
Using a Network Drive to Deploy Test Systems	982
Using a Source Code Control System to Deploy Test Systems	985
Preparing Source Components for Deployment	988
Deploying LabVIEW Project Libraries	993
Partial Project Libraries	994
Troubleshooting LabVIEW Code Module Issues	996
Using Multiple LabVIEW Versions in a Test System	998
Customizing Components You Deploy	1000
Configuration Files	1004
User Components	1010
Deploying Type Palette Files	1011
Creating Deployments	1012
Using a TestStand Workspace File to Create a Deployment	1013
Using a Directory to Create a Deployment	1015
Analyzing the Test System	1015
Including Source Components	1017
Including Files from the TestStand Public Directory in a Deployment...	1017
Including a User Interface in a Deployment	1018
Including TestStand Configuration Files in a Deployment	1019
Including Additional NI Installers in a Deployment	1019
Including Hardware Configuration Information in a Deployment .	1021
Manually Adding or Removing Files to or from Deployments	1022
Filtering and Excluding Files	1023
Processing Files for Deployment	1028
Processing LabVIEW Code Modules for Deployment	1028
Processing LabVIEW NXG Code Modules for Deployment	1030
Processing TestStand Sequence Files for Deployment	1032
Processing .NET Code Modules for Deployment	1033
Creating a Deployable Image	1033
Choosing Single or Multiple Deployments	1038
Using a Dedicated Build Computer	1040
Building a Customized MSI-based Installer	1041
Relationship of NI MSI-based Installer Architecture and Microsoft	

Windows Installer Technology	1044
Microsoft Windows Merge Modules	1045
NI MSI-based Installer Architecture	1046
Redistributing MSI-based Installer Products in a Deployment.....	1047
Using Custom Commands to Execute Third-Party Installers	1049
Clearing the Product Cache	1050
Managing Versioned and Non-Versioned Files	1051
Upgrade Codes That Affect MSI-based Installer Behavior.....	1052
Building a Customized Package Distribution.....	1053
Patching Deployments	1055
Differences between Patch Deployments and Full Deployments ..	1058
Limitations of Patch Deployments	1061
Creating Independent Patch Deployments	1063
Including and Excluding Files in MSI-based Installer Patches	1064
Adding New Files to a Patch Deployment.....	1064
Including Modified Files in a Patch Deployment.....	1065
Including Unmodified Files in a Patch Deployment.....	1065
Including NI Drivers or the TestStand Runtime in a Patch Deployment and Installer.....	1066
Excluding New or Modified Files from a Patch Deployment ..	1066
Differences between Standard Mode and Manual Mode.....	1067
ADE-specific Issues for MSI-based Installer Patches.....	1069
Patching LabVIEW VIs.....	1069
Patching LabWindows/CVI DLLs and Executables.....	1070
Patching Microsoft .NET Assemblies	1070
Debugging a Patch Deployment	1071
Patching Custom Sequence Editors	1072
Reverting a Patch Deployment	1072
Transferring Files	1073
Deploying Multiple Test Systems to the Same Test Station	1074
Validating Deployments	1078
Common Deployment Scenarios	1081
Deploying the TestStand Runtime	1082
Distributing Tests from a Workspace	1084
Adding Dynamically Called Files to a Workspace	1086

Distributing a User Interface	1089
Improving Performance for Creating Deployments	1091
Automating Deployment with TestStand Deployment Utility	1093
Effectively Using LabVIEW with TestStand.	1100
Organizing LabVIEW-Based TestStand Systems	1100
Organizing Test Program Files with LabVIEW Projects.	1101
Editing VIs in LabVIEW Projects.	1102
Updating and Patching VIs in LabVIEW Projects	1103
Deploying VIs in LabVIEW Projects	1104
Organizing Test Program Files with LabVIEW Packed Project Libraries	1104
Editing VIs in LabVIEW Packed Project Libraries	1105
Updating and Patching VIs in LabVIEW Packed Project Libraries	1107
Deploying VIs in LabVIEW Packed Project Libraries	1107
Organizing Test Program Files with Stand-alone VIs	1108
Editing Stand-alone VIs.	1109
Updating and Patching Stand-alone VIs	1110
Deploying Stand-alone VIs	1111
Organizing Test Program Files with LabVIEW Libraries	1112
Organizing Test Program Files with LabVIEW Project Libraries	1113
Organizing Test Program Files with LabVIEW-Built Shared Libraries	1114
Special Considerations for Using LabVIEW with TestStand Systems	1115
Managing Memory Allocation for Large LabVIEW Data Sets	1116
Reusing VIs.	1116
Reserving Loaded VIs for Execution	1117
Partially Specifying LabVIEW Clusters	1118
Executing VIs with Separate Compiled Code	1119
Evaluating Conditional Disable Structures and Symbols	1119
Calling VIs That Create ActiveX References.	1120
Calling VIs That Use XControls	1120
Building a TestStand Deployment with LabVIEW	1120
Creating Custom Step Types that Call VIs for Substeps.	1121
Calling LabVIEW Class Member VIs from TestStand	1122
Passing an Instance of a LabVIEW Class Object.	1123
Calling LabVIEW Property Nodes from TestStand.	1125
Debugging and Tracing of VIs Executed Using the LabVIEW Runtime	1126
Debugging Code Module VIs with LabVIEW	1127

Tracing with the DETT	1130
TCP Ports for the LabVIEW Runtimes	1131
Deploying Network-Published Shared Variables	1132
Resolving References to Shared Variables	1134
Symmetric Multiprocessing in VIs Executed from TestStand	1135
Configuring the Number of LabVIEW RTE Execution Threads	1137
Configuring the CPU Affinity for LabVIEW Execution Threads	1137
General Strategies for Optimizing LabVIEW Code Module Performance on SMP Systems	1139
Configuring the Number of Execution Threads Assigned to a LabVIEW Execution System	1140
Calling VIs on Remote Computers	1142
Configuring a LabVIEW Step to Run Remotely	1143
Configuring the LabVIEW VI Server to Run VIs Remotely	1145
TestStand Example Programs	1147
Examples for Built-In Step Types	1147
Database Step Types	1147
Flow Control Step Types	1149
IVI Step Types	1153
Message Popup Step Type - Automatically Dismissing	1155
Message Popup Step Type - Get User Input	1157
Multiple Numeric Limit Test Step Type	1159
Multiple Numeric Limit Test Step Type - LabWindows/CVI	1159
Multiple Numeric Limit Test Step Type - LabVIEW	1161
Multiple Numeric Limit Test Step Type - .NET	1163
Property Loader Step Type - Loading Limits from a Database	1165
Property Loader Step Type - Loading Limits from a File Containing Alias	1168
Property Loader Step Type - Loading Limits from a Microsoft Excel File	1169
Property Loader Step Type - Loading Limits for Multiple Sequences	1171
Property Loader Step Type - Loading Limits from Multiple Sources	1172
Property Loader Step Type - Loading Limits from Two Different Text Files ...	1174
Statement Step Type	1176
Synchronization Step Types - Auto Schedule	1177
Synchronization Step Types - Batch Synchronization	1179
Synchronization Step Types - Lock	1181
Synchronization Step Types - Notification and Wait	1183

Synchronization Step Types - Queue	1184
Synchronization Step Types - Rendezvous and Semaphore	1186
Examples for Custom Step Types	1188
Agilent VEE Step Types	1189
ATLAS Step Type	1191
Edit ATLAS Call Dialog Box	1196
Error Codes and Descriptions	1198
Java Step Types	1199
Edit Java Call Dialog Box	1204
Error Codes and Descriptions	1206
Examples for Customizing Result Processing	1208
Adding Custom Data to a Report	1208
Adding Custom Images to a Report	1210
Displaying Graphs in a Report	1212
Including Hyperlinks in a Report	1214
Including Hyperlinks in a Report - TDMS File	1215
Model Plug-in - Basic Step Time Report	1216
Model Plug-in - Simple Text Report	1223
Examples for Demos	1226
Computer Motherboard Test	1226
Computer Motherboard Test Sequence - HTBasic	1226
Computer Motherboard Test Sequence - LabWindows/CVI	1228
Computer Motherboard Test Sequence - LabVIEW	1230
Computer Motherboard Test Sequence - .NET	1232
Computer Motherboard Test Sequence - Python	1234
Mobile Device Test Demo	1236
Parallel Test Strategies Demo	1238
Examples for Fundamentals	1240
Calling Dynamic Dispatch VIs	1240
Developing Platform Independent Test Systems	1241
Launching a Modal Dialog	1243
Launching a Modal Dialog - LabVIEW	1243
Launching a Modal Dialog - MFC	1245
Launching a Modal Dialog - .NET	1246
Launching an MFC Dialog with ActiveX Controls	1247
Overriding Engine Callbacks - SequenceFilePostStepFailure	1248

Overriding Engine Callbacks - SequenceFilePostStepRuntimeError	1250
Passing Clusters to Code Modules - LabVIEW	1252
Passing Structs to Code Modules	1254
Passing Structs to Code Modules - C	1255
Passing Structs to Code Modules - LabWindows/CVI	1257
Passing Structs to Code Modules - LabVIEW DLL	1260
Passing Structs to Code Modules - .NET	1262
Pausing Executions with Running Code Modules	1264
Pausing Executions with Running Code Modules - LabWindows/CVI	1264
Pausing Executions with Running Code Modules - LabVIEW	1266
Pausing Executions with Running Code Modules - .NET	1267
Interpreter Session Management in Python	1269
Passing Data between TestStand and Python	1271
Sequence File Translators	1276
Structure of TestStand Executions	1278
Termination Monitor	1279
Termination Monitor - LabWindows/CVI	1280
Termination Monitor - LabVIEW	1281
Termination Monitor - .NET	1283
Using Data Streams	1284
Using Data Streams - LabVIEW	1285
Using Data Streams - .NET	1286
Using Sweep Loops	1286
Using Sweep Loops - LabVIEW	1287
Using Sweep Loops - .NET	1288
Sweep Loop Strategies	1288
Using TestStand Enumerations	1289
Using TestStand Enumerations - CVI	1290
Using TestStand Enumerations - LabVIEW	1291
Using TestStand Enumerations - .NET	1293
Examples for Interfacing with Hardware	1294
Instrument Control Step Types	1295
InstrumentStudio	1301
Measure Efficiency	1301
Measure Quiescent Current	1303
Interfacing with NI-Hardware drivers using Python Custom step and Python	

Adapter	1306
Session Manager	1309
Session Manager - LabWindows/CVI	1309
Session Manager - LabVIEW	1311
Switch Executive	1313
Characterization of a Transistor	1314
Examples for Modifying Process Models	1316
Dynamically Set the Client Sequence File	1316
Overriding PreUUT Model Callbacks	1319
Overriding PreUUT and PostUUT Parallel Model Callbacks	1320
Overriding PreUUT and PostUUT Batch Model Callbacks	1322
Overriding PreBatch and PostBatch Model Callbacks	1323
Examples for Modifying User Interfaces	1325
Building a TestStand UI with Native Controls	1325
Building a TestStand UI with Native Controls - LabWindows/CVI	1325
Building a TestStand UI with Native Controls - LabVIEW	1327
Building a TestStand UI with Native Controls - .NET	1328
Connecting UI Controls Demo	1330
Creating a Basic User Interface	1331
Creating a Basic User Interface - LabWindows/CVI	1332
Creating a Basic User Interface - LabVIEW	1333
Creating a Basic User Interface - .NET	1335
Handling UI Messages	1336
Handling UI Messages - LabWindows/CVI	1337
Handling UI Messages - LabVIEW	1339
Handling UI Messages - .NET	1341
Updating the Status Bar Using UI Messages	1343
Updating the Status Bar Using UI Messages - LabWindows/CVI	1344
Updating the Status Bar Using UI Messages - LabVIEW	1345
Updating the Status Bar Using UI Messages - MFC	1347
Updating the Status Bar Using UI Messages - .NET	1349
Examples for Parallel Testing	1350
Comparing Resource Usage Strategies	1350
Executing Code Modules in Parallel	1355
Executing Sequences in Parallel	1357
Testing UUTs in Parallel - Batch Model	1358

Testing UUTs in Parallel - Parallel Model	1360
Examples for the TestStand API	1362
Accessing Arrays Using API	1362
Accessing Arrays Using API - HTBasic	1362
Accessing Arrays Using API - LabWindows/CVI	1364
Accessing Arrays Using API - LabVIEW	1366
Accessing Arrays Using API - MFC	1368
Accessing Arrays Using API - .NET	1370
Accessing Properties Using API	1371
Accessing Properties Using API - LabWindows/CVI	1372
Accessing Properties Using API - LabVIEW	1373
Accessing Properties Using API - MFC	1375
Accessing Properties Using API - .NET	1377
Accessing Properties Using API - TestStand Expressions	1379
Building a Sequence Using API	1381
Building a Sequence Using API - LabVIEW	1381
Building a Sequence Using API - .NET	1382
Building a Sequence Using API - TestStand Expressions (INI)	1384
Building a Sequence Using API - TestStand Expressions (XML)	1385
Creating and Deleting Users Using API	1386
Creating New Properties Using API	1389
Creating New Properties Using API - LabWindows/CVI	1389
Creating New Properties Using API - LabVIEW	1391
Creating New Properties Using API - .NET	1394
Creating New Properties Using API - TestStand Expressions	1396
Executing Sequences Using API	1398
Executing Sequences Using API - LabWindows/CVI	1398
Executing Sequences Using API - LabVIEW	1401
Executing Sequences Using API - .NET	1403
Executing Sequences Using API - TestStand Expressions	1405
Type Casting API Classes	1407
Type Casting API Classes - LabWindows/CVI	1407
Type Casting API Classes - LabVIEW	1409
Type Casting API Classes - .NET	1410
Type Casting API Classes - TestStand Expressions	1412
Examples for TestStand Debugging Features	1413

Benchmarks	1413
Custom Analyzer Rules	1415
Generating Output Messages	1417
TestStand API Overview.....	1420
TestStand ActiveX API Overview	1420
ActiveX Automation Server Concepts	1420
Objects, Classes, Methods, Properties, and Controls.....	1421
Object References	1421
Adding and Releasing References.....	1422
Releasing ActiveX References in LabVIEW.....	1423
Adding and Releasing References in LabWindows/CVI	1423
Adding and Releasing References in Microsoft Visual Basic .NET ..	1425
Adding and Releasing References in C#.....	1426
Adding and Releasing References in Microsoft Visual	
C++/#import	1426
TestStand API Concepts	1427
Accessing Built-in Properties.....	1427
Accessing Dynamic Properties	1428
Acquiring a Derived Class from the PropertyObject Class	1428
Object Relationships	1429
API Inheritance	1429
API Containment	1430
Property Paths	1430
Finding a Property Path	1431
Lookup Strings	1432
Thread Concurrency Models	1433
Programming with the TestStand API in Different Languages.....	1434
Programming with the TestStand API in LabVIEW	1435
Invoking Methods in LabVIEW	1435
Accessing Built-in Properties in LabVIEW	1435
Accessing Dynamic Properties in LabVIEW.....	1436
Programming with TestStand API Constants and Enumerations in	
LabVIEW	1437
Obtaining a Different Interface for TestStand Objects in LabVIEW .	1438
Acquiring a Derived Class from the PropertyObject Class in LabVIEW...	

1439	
Duplicating COM References in LabVIEW	1440
Setting the Preferred Execution System of LabVIEW VIs to Prevent Deadlock	1441
Programming with the TestStand API in LabWindows/CVI.....	1442
Using ActiveX Drivers in LabWindows/CVI	1442
Invoking Methods in LabWindows/CVI	1443
Adding and Releasing References in LabWindows/CVI	1445
Accessing Built-in Properties in LabWindows/CVI	1447
Accessing Dynamic Properties in LabWindows/CVI	1449
Programming with TestStand API Constants and Enumerations in LabWindows/CVI	1450
Programming with the TestStand API in Microsoft Visual Basic .NET....	1451
Accessing Built-in Properties in Microsoft Visual Basic .NET	1452
Accessing Dynamic Properties in Microsoft Visual Basic .NET	1453
Acquiring a Derived Class from the PropertyObject Class in Microsoft Visual Basic .NET	1454
Programming with the TestStand API in C#	1454
Accessing Built-in Properties in C#.....	1456
Accessing Dynamic Properties in C#	1457
Acquiring a Derived Class from the PropertyObject Class in C# ...	1458
Programming with the TestStand API in Microsoft Visual C++	1459
Object References in Microsoft Visual C++/#import	1461
Strings in Microsoft Visual C++/#import	1461
Utilities for Microsoft Visual C++/#import.....	1462
Accessing Built-in Properties in Microsoft Visual C++/#import	1465
Accessing Dynamic Properties in Microsoft Visual C++/#import ...	1465
Acquiring a Derived Class from the PropertyObject Class in Microsoft Visual C++/#import	1466
Programming with the TestStand API in HTBasic.....	1467
.NET Support for the TestStand API	1469
Writing an Application with the TestStand User Interface Controls	1469
Writing an Application with the TestStand Engine API	1470
Creating the TestStand Engine	1470
User Interface Messages (UIMessages)	1470
Creating a User Interface Message Handler	1471

Handling Specific User Interface Messages	1472
Execution Events	1472
Shutdown Events	1475
User-Defined Events	1476
Handling Unknown Events	1476
Posting User Interface Messages	1476
Managing Sequence Files in an Application	1477
Managing Process Model Entry Points	1479
Implementing a Tools Menu	1483
Executing a Sequence in an Application	1483
Starting an Interactive Execution in an Application	1484
Log in/Log out Execution	1485
Shutting Down the Engine	1487
Enforcing User Privileges	1489
Resource Strings and Internationalization	1489
Thread Safety of the PropertyObject API and TestStand Variables	1490

TestStand User Manual

The TestStand User Manual provides detailed descriptions of the product functionality and the step by step processes for use.

Looking for Something Else?

For information not found in the User Manual for your product, such as specifications and API reference, browse ***Related Information***.

Related concepts:

- [Activating and Licensing TestStand](#)

Related tasks:

- [Installing TestStand](#)

Related information:

- [TestStand Release Notes - Known Issues, Bug Fixes, and More](#)
- [TestStand API Reference](#)

TestStand Overview

TestStand is test management software that helps you develop, debug, and deploy test systems and provides full visibility into testing processes and results. TestStand helps engineers quickly develop robust automated test and validation systems.

Related information:

- [TestStand Licensing Options](#)
- [License Setup and Activation](#)

TestStand Key Features

- Develop test sequences that integrate code modules written in any programming language
- Profile and optimize speed and parallelism before deploying test systems to production
- Meet the needs of any environment with extensible plug-ins for reporting, database logging, and connectivity to other systems
- Deploy test systems with increased throughput that run at production speed

Components of a TestStand System

TestStand is designed for use in a test system along with hardware, drivers, and additional software to optimize TestStand for your application. Use the following list of typical TestStand system components as a starting point for building your test system.

Table 1. Components of a TestStand System

Component	Notes
TestStand	Install TestStand using NI Package Manager. TestStand includes the TestStand engine, sequence editor, and user interface.

Component	Notes
Code Modules	Code modules are programs invoked by TestStand. Create code modules in the supported application development environment (ADE) or programming language of your choice.
Module Adapters	Module adapters invoke code modules that your TestStand sequences call. TestStand installs several module adapters including, LabVIEW, .NET, and Python. This allows you to load and call code modules, pass parameters to code modules, and return values and status from code modules you created in a variety of programming languages.
Hardware and Drivers	The hardware and drivers required for your system depend on the device, component, and properties you want to test. Visit <i>ni.com</i> to explore the ecosystem of NI test and measurement hardware.

Related concepts:

- [Parts of a TestStand Application](#)
- [Module Adapters](#)

Related information:

- [Installing, Updating, Repairing, and Removing NI Software](#)

Installing TestStand

Consider the following conditions when installing TestStand on your system:

- The default TestStand installation location is `<Program Files>\National Instruments\TestStand <Version Year>`.
- You can install TestStand on a computer which has a previous TestStand version installed, but you cannot install TestStand over a previous version and cannot install TestStand in the default installation directory for a previous version. If you installed a previous version of TestStand in a non-default directory, you can uninstall the previous version of TestStand and install the new version of TestStand to that directory.
- You cannot install TestStand to a network path or a mapped network path. You must install TestStand on a local computer.
- If you uninstall one bitness of a TestStand version, NI recommends that you use the TestStand Version Selector to reactivate the remaining bitness of that version and launch TestStand to properly register its components. For example, if you uninstall TestStand 2023 Q4 64-bit, use the TestStand Version Selector to set TestStand 2023 Q4 32-bit to active and launch TestStand to properly register your remaining version of TestStand 2023 Q4.

1. Download the TestStand installer.



Note If you purchased this product with an NI Software Suite or NI Product Bundle, use the installation media that shipped with your purchase to install this product.

2. Run the TestStand installer and follow instructions to select your desired software components and install them. Consider the following information for unique installation configurations:
 - To install only TestStand Runtime on a machine, deselect all optional products by clicking **Deselect All** on the **Select** tab. You cannot automate the installation of TestStand Runtime in NI Package Manager using the command line. To automate the installation of TestStand Runtime, you must create a custom installer using NI Package Builder.
 - The TestStand `GraphControl` and `IVISepTypes` packages are available

in the TestStand offline installer for manual or automatic installation using NI Package Manager. These packages can be included in custom installers built using NI Package Builder.

- TestStand can use newer versions of the LabVIEW Runtime Engine (RTE) when you install LabVIEW on a development system. You can include newer versions of the LabVIEW RTE in deployments using the Drivers and Components dialog box of the TestStand Deployment Utility.



Note NI Package Manager is required to install TestStand. The installer will install NI Package Manager first if it is not present on your machine.

3. Reboot your system once the installation is complete.

Activate TestStand using one of the following activation methods:

- Log in to your NI user account to check for associated licenses
- Enter the serial number on the Certificate of Ownership included with your TestStand software kit
- Enter an activation code
- Connect to a volume license server

Related information:

- [Download TestStand](#)
- [NI Package Manager Manual](#)
- [Downloading Individual Offline Installers for NI Software](#)
- [NI Package Builder Manual](#)
- [Activating Software](#)
- [License Setup and Activation](#)

Activating and Licensing TestStand

After you install TestStand, you must use NI License Manager to activate the software or initiate the evaluation period for the software.

Activate TestStand using one of the following activation methods:

- Log in to your NI user account to check for associated licenses
- Enter the serial number on the Certificate of Ownership included with your TestStand software kit
- Enter an activation code
- Connect to a volume license server

NI offers a variety of licenses for the different ways you can use TestStand in development and deployment applications. You can select from the following types of licenses:

- TestStand Development System License
- TestStand Continuous Integration License
- TestStand Custom Sequence Editor License
- TestStand Debug Deployment Environment License
- TestStand Base Deployment Engine License

Use the following descriptions only as a reference for the licensing options. Refer to ni.com/activate for more information about activating TestStand licenses. Refer to ni.com/teststand to purchase a TestStand license. Contact a local NI representative for more information or for questions about specific licensing needs.



Note This document does not replace the NI Software License Agreement installed in the <National Instruments>\Shared\MDF\EULAs\NIReleased directory.

Table 2. TestStand License Part Numbers

License Name	Part Number
TestStand Development System License	788372-35

License Name	Part Number
TestStand Continuous Integration License for VLAs	790410-35
TestStand Custom Sequence Editor License	777775-35
TestStand Debug Deployment Environment License	779851-35
TestStand Base Deployment Engine License	777774-35

TestStand Evaluation Package

When you run TestStand in Evaluation Mode, the software expires after 7 days. The software runs as a fully functional Development System during the evaluation period. You can use an ni.com user profile to extend the evaluation period for an additional 45 days. You can activate a license at any point during or after the evaluation period.

TestStand Development System License

Activate this license to develop and edit test sequences and to save sequence files within the TestStand Sequence Editor. This license also allows you to develop and edit test sequences within a TestStand Custom Sequence Editor or programmatically using the TestStand API. This license also grants the right to develop Custom Sequence Editors and User Interfaces. You must have an ni.com user profile to activate a TestStand Development System license. The TestStand Development System license allows you to:

- Design, debug, and develop sequences
- Develop a User Interface
- Create a deployment
- Develop a Custom Sequence Editor
- Use TestStand on an additional machine or container within a continuous integration (CI) or continuous development (CD) workflow

TestStand Continuous Integration License

Use the TestStand Continuous Integration license to include TestStand in your CI/CD workflows at no additional cost.

Single-Seat License Users

If you have an individual TestStand Development System license, you can use TestStand on an additional machine or container for CI/CD workflows such as automated software building. Enterprise agreements also allow you to use TestStand for CI/CD workflows.

Volume License Agreement Users

If you have a Volume License Agreement (VLA) with NI which includes a TestStand Development System license, you can use TestStand in CI/CD workflows without using a development license seat. Add this support by adding the ***TestStand Continuous Integration for VLAs*** part number, 790410-35, to your VLA. This part number is free to add to your VLA and gives you a TestStand license to use on your CI/CD machine.



Note The ***TestStand Continuous Integration for VLAs*** part number does not grant the right to perform any development tasks, such as editing application functionality, using TestStand.

TestStand Custom Sequence Editor License

Activate this license to deploy a sequence editor customized to your needs in a variety of languages and platforms, including LabVIEW, LabWindows/CVI, C++, and C#. This license also allows you to develop and edit test sequences within a TestStand Custom Sequence Editor or programmatically using the TestStand API.



Note TestStand Custom Sequence Editor licenses are necessary on any machines where a test system that includes a Custom Sequence Editor is deployed.

The TestStand Custom Sequence Editor license allows you to:

- Design, debug, and develop sequences within a Custom Sequence Editor
- Create a deployment

TestStand Debug Deployment Environment License

Activate this license to install the development versions of TestStand, LabVIEW, LabWindows/CVI, MeasurementStudio, and any corresponding add-on toolkits on a

single test station so you can debug deployed test applications on the test station. This license grants the right to make minor edits to fix bugs in deployed test applications but does not grant the right to perform any development tasks, such as adding or improving application functionality, using TestStand, LabVIEW, LabWindows/CVI, or MeasurementStudio on the test station.

You cannot activate and deactivate the TestStand Debug Deployment Environment license and reuse it on multiple computers. If you need to use a single debug license across multiple computers, contact NI for more information about the Concurrent TestStand Debug Deployment Environment license.

The TestStand Debug Deployment Environment license allows you to:

- Debug test sequences on a deployed machine

TestStand Base Deployment Engine License

Activate this license on a deployment machine to execute TestStand deployments. This license does not grant the right to perform any development tasks using the TestStand Sequence Editor, a TestStand Custom Sequence Editor, or the TestStand API.



Note The TestStand Base Deployment Engine license is the minimum license required for all deployed TestStand-based applications.

The TestStand Base Deployment Engine license allows you to:

- Execute TestStand deployments

If you upgrade TestStand with a service pack release by installing TestStand or a deployment that includes the TestStand Engine, the TestStand Engine on the computer becomes unlicensed. You must reactivate the license on the computer. TestStand licenses tied to a Standard Service Program (SSP) exhibit the following behavior when reactivated:

License	Behavior
TestStand Development System License	The license must be current with the SSP to reactivate a service pack license.
TestStand Debug Deployment Environment	The license must be current with the SSP to

License	Behavior
License	reactivate a service pack license.
TestStand Custom Sequence Editor License	You can reactivate the license for a service pack upgrade even if the SSP for the original license lapsed.
TestStand Base Deployment Engine License	Does not require SSP to reactivate a service pack license.

TestStand Tools That Do Not Require Activation

The following tools do not require activation for build or analysis purposes, as defined by the license agreement:

- TestStand Deployment Utility
- Update VI Calls Application
- TestStand Sequence Analyzer
- Offline Results Processing Utility
- File Diff and Merge Utility (You can diff, but not save, sequence files)
- XML File Packaging Utility
- Database Viewer

Related information:

- [License Setup and Activation](#)
- [Activating Software](#)
- [Purchase a TestStand License](#)

TestStand New Features and Changes

Learn about updates, including new features and behavior changes, introduced in each version of TestStand.

Discover what is new in the latest releases of TestStand.



Note If you cannot find new features and changes for your version, it might not include user-facing updates. However, your version might include non-visible changes such as bug fixes and compatibility updates. For information about non-visible changes, refer to your product **Release Notes**.

Related information:

- [TestStand Release Notes - Known Issues, Bug Fixes, and More](#)

TestStand 2025 Q3 Changes

Learn about new features, behavior changes, and other updates in TestStand 2025 Q3.

Introducing Nigel™, AI Advisor

Nigel is your trusted test and measurement advisor, trained on the NI platform and ready to help when you need it. Change TestStand and sequence file settings, learn TestStand concepts, and more!

Import Specifications Using the Specifications Pane

Use the Specifications pane to import specifications directly from a SystemLink Enterprise server or Excel file and convert the imported specifications into steps, sweep loops, and variables in your sequence files. This functionality replaces the Specification Compliance Manager (SCM) tool specification import workflow.

Related concepts:

- [Importing Specifications into TestStand Using the Specifications Pane](#)

- [Nigel, AI Advisor](#)

TestStand 2025 Q2 Changes

Learn about new features, behavior changes, and other updates in TestStand 2025 Q2.

TestStand 2025 Q2 Installation Location

If you install TestStand 2025 Q2 on a device where TestStand 2024 Q4 is already installed, TestStand 2025 Q2 installs as an update to TestStand 2024 Q4 rather than a separate instance of TestStand software. Installing TestStand 2025 Q2 on a device where no prior TestStand software is installed, or on a device where an older version of TestStand than 2024 Q4 is installed, installs TestStand 2025 Q2 as a separate software instance at `C:\<Program Files>\National Instruments\TestStand 2024`.

.NET 8 Support Updates in TestStand

All .NET-based tools, plug-ins, and interfaces in TestStand have been migrated to .NET 8, which allows you to call into .NET code modules from TestStand. This includes the following items:

- TestStand Sequence Editor
- TestStand user interfaces and operator interfaces
- Tools such as the TestStand Sequence File Analyzer and TestStand File Diff and Merge Utility
- Step settings panels and substeps of all TestStand step types
- .NET-based plug-ins such as process models and the TestStand Sequence Analyzer



Note If you are using deprecated APIs or features no longer supported in .NET 8, you might need to upgrade to .NET core or make changes to your existing modules to retain functionality.

Git Pane

The TestStand Sequence Editor provides a Git pane that allows you to connect to Git repositories on platforms including Azure DevOps, GitHub, and BitBucket and perform

common version control operations from within the Sequence Editor. This new pane eliminates the need to perform version control operations on TestStand project files using a separate Git client.

TestStand Usage Within Continuous Integration Workflows

TestStand Development licenses now support the use of TestStand within continuous integration (CI) and continuous development (CD) workflows. Single-seat license users can now use their TestStand Development System licenses for both development purposes and CI/CD workflows, while volume license agreement (VLA) users can add the ***TestStand Continuous Integration for VLAs*** part number to their VLA to use TestStand in CI/CD workflows without using a development license seat. The TestStand End-User License Agreement (EULA) supports CI/CD uses of the TestStand Development System license.

API and UI Control Additions and Changes

TestStand 2025 Q2 includes the following new enumeration values, constant values, properties, and methods.

- Properties
 - `DotNetModule.AllowUnload`
- Methods
 - `DotNetAdapter.CacheAssemblyInfoEx`
 - `DotNetAdapter.IsClassValidEx`



Note Although some TestStand components, such as the TestStand Sequence Editor, expose .NET assemblies, TestStand does not support the use of undocumented .NET APIs.

Related concepts:

- [Selecting a Source Code Control Provider](#)
- [Activating and Licensing TestStand](#)
- [Managed Code Considerations for TestStand and .NET 8](#)
- [Calling LabVIEW VIs That Invoke .NET Code in TestStand](#)

Related reference:

- [Migrating User Interfaces to .NET Core](#)

TestStand 2024 Q4 Changes

Learn about new features, behavior changes, and other updates in TestStand 2024 Q4.

.NET Adapter — .NET Assembly Execution Support

The .NET adapter now supports .NET 8 for .NET code module execution. You can still execute .NET Framework assemblies using .NET as long as the resources used in the .NET Framework assemblies are present in .NET 8.

Some items and resources are no longer supported. Consult the following list for some support changes and considerations to take note of when developing with the .NET Adapter and TestStand 2024 Q4:

- The `IOConfig` step type no longer works with the .NET Adapter. The device drivers are located in the .NET Framework GAC location, and .NET 8 does not support loading assemblies from the GAC.
- Creating remote objects in the TestStand .NET step using the `CreateRemoteObject` method is no longer supported.
- Calling into the following assemblies is no longer supported:
 - GAC assemblies
 - .NET assemblies with dependencies on GAC assemblies
 - C++/CLI assemblies
- Consider the following when working with .NET assemblies and LabVIEW:
 - .NET assemblies called by a VI are not automatically added to TestStand deployments. You must manually add these dependencies to deployments they are used in.
 - NET Framework assembly dependencies called by LabVIEW during the execution of a TestStand .NET step must be placed next to these .NET Framework assemblies to prevent runtime execution errors. These dependencies can be found in the LabVIEW installation directory.
 - Passing .NET objects between LabVIEW and TestStand is no longer supported.
- Sequence Editor layout files from previous TestStand versions are no longer

supported.

- Consider the following when handling TestStand objects in managed code:
 - Explicitly dispose of instances of classes implementing `IDisposable`.
 - Explicitly assign `null` to class member variables of TestStand types, like `Engine`, `SequenceContext`, and `PropertyObject`, when the class is no longer in use.
 - Explicitly assign `null` to static variables of TestStand types, like `Engine`, `SequenceContext`, and `PropertyObject`, before shutting the TestStand Engine down.

Support for Using Sequence Files Across Multiple Versions of TestStand

The TestStand Engine no longer updates to a new version with every new release of TestStand. This allows you to use sequence files in multiple versions of TestStand without needing to use **Save As** to save the file for each version of TestStand you want to use it in. **Save As** now only needs to be used to run sequence files in versions of TestStand with different engine versions than the version it was originally written in. The engine version of your version of TestStand can be determined by consulting the program information at **Help » About TestStand**.

Features from later versions of TestStand are not supported in earlier versions of TestStand, even if the versions run on the same engine version. The TestStand Analyzer can detect these support mismatches and notify the user.

Support mismatches can also occur in custom types used between versions of TestStand. You can update your custom types according to the recommendations in ***Modifying Types Without Altering the Version Number*** to allow the TestStand Analyzer to detect and report support mismatches in custom data types.

Python Adapter — Support for Python Virtual Environments Created Using 'venv'

The TestStand Python Adapter now supports Python virtual environments created using the `venv` tool, in addition to virtual environments created using the earlier `virtualenv` tool.

LabVIEW Adapter — Source-Only VI Execution Support Through the Version-Independent Runtime Engine Support Feature

You can use the **Version Independent Runtime Engine Support** feature in the LabVIEW Adapter as an option to avoid potential issues when executing source-only VIs using the LabVIEW Runtime Engine through the LabVIEW adapter. These issues can occur if the LabVIEW version that the source-only VI was saved or mass-compiled in does not match the LabVIEW Runtime engine version, as VI version no longer gets updated during a save or mass-compile.

Auto Login for TestStand Default User

The default TestStand user account, `administrator`, no longer launches the login dialog prompting users to log in when launching TestStand. The login dialog only launches if at least one of the following is true:

- A user clicks on the **Login** option
- The `administrator` account has a configured password
- Additional user accounts other than `administrator` are present

TestStand Sequence Editor Visual Design Updates

TestStand 2024 Q4 introduces visual changes to the TestStand Sequence Editor designed to provide better contrast in the Sequence Editor, provide relief in the menu and tool bars, and reduce the background colour of Sequence Editor panes.

New Help Delivery Experience

TestStand 2024 Q4 help links now direct you to the online TestStand help documentation in the NI documentation portal, ***ni.com/docs***, when your system is connected to the internet. If your system is not connected to the internet, the help links launch the NI Offline Help Viewer that is installed with TestStand to connect you with help documentation. You can also use the NI Offline Help Viewer to launch TestStand help documentation from outside of TestStand.



Note Use the NI Help Preferences application to set the default method for accessing NI product help to online or offline.



Note TestStand 2024 Q4 removes offline help documentation sourced in the Microsoft Compiled HTML Help File (.chm) format.

API and UI Control Additions and Changes

TestStand 2024 Q4 includes the following new enumeration values, constant values, properties, and methods.

- Enumeration and Constant Values
 - `StepType_If`
 - `StepType_Else`
 - `StepType_ElseIf`
 - `StepType_For`
 - `StepType_ForEach`
 - `StepType_While`
 - `StepType_DoWhile`
 - `StepType_Break`
 - `StepType_Continue`
 - `StepType_Select`
 - `StepType_Case`
 - `StepType_End`
 - `StepType_StreamLoop`
 - `StepType_SweepLoop`
 - `StepType_Wait`
 - `VisualStudioDTEVersion_2022`
- Properties
 - `EngineVersionString`
 - `ChildExecutionIds`
 - `ParentExecutionId`
- Methods
 - `DisplayHelpTopicEx`
 - `GetExecution`



Note Although some TestStand components, such as the TestStand Sequence Editor, expose .NET assemblies, TestStand does not support the use of undocumented .NET APIs.

Related concepts:

- [Modifying Types Without Altering the Version Number](#)

Related tasks:

- [Executing Source-Only VIs in LabVIEW Runtime Engine](#)

Related information:

- [Microsoft - Breaking Changes for Migration from .NET Framework to .NET Core](#)

TestStand 2023 Q4 Changes

Learn about new features, behavior changes, and other updates in TestStand 2023 Q4.

Build Source Files and Execute Feature

The new **Build Source Files and Execute** feature adds support for executing source-only VIs in TestStand using the TestStand LabVIEW adapter when it is configured to use the LabVIEW runtime engine. This feature adds this support by compiling LabVIEW modules necessary for execution into a Packed Project Library (PPL) before execution begins.

Import/Update from Specification Compliance Manager Tool

The new Import/Update from Specification Compliance Manager tool lets you connect to a Specification Compliance Manager (SCM) server from TestStand. You can import data from a server to a sequence file as variables to use in configuring measurement conditions, sweeps, and test limits. The tool also allows you to log results tied to imported data and pass the logged results back to SCM for analysis.

TestStand Installer Feed Optimization

Reduced TestStand ISO image size improves download times and decreases the necessary storage space for the ISO image. Installers for GraphControl, IVIStepTypes, Runtime32, and Runtime64 have been removed from the ISO image, but are still

available for installation through NI Package Manager and NI Package Builder.

Improvements to Working with Variables

You can now filter variables and their properties by name in the Call Stack pane using the new **Filter** control on the Variables pane on the Sequence File tab.

Hot Reloading in Visual Studio 2022 for C# and C++ Code Modules

TestStand now supports modifying C# and C++ code modules during debugging sessions in Visual Studio (VS) 2022 without the need to reload the modules in TestStand to get the updated code modules reloaded to memory. This added functionality leverages the hot reloading capability in VS 2022.

Python Adapter Improvements

Improved TestStand performance when using the `Enum` data type with the TestStand Python adapter to pass `Enum` values between TestStand and Python code modules.

API and UI Control Additions and Changes

TestStand 2023 Q4 includes the following new enumeration values, constant values, properties, and methods.



Note This list may not be complete.

- Enumeration and Constant Values
 - `UIStyleFontType_Subtitle2`
- Properties
 - `ApplicationVersionString`
 - `BuildAndExecuteAtStartOfExecution`
- Methods
 - `ClearBuildAndExecuteCache`
 - `IsBuildAndExecuteEnabled`



Note Although some TestStand components, such as the TestStand

Sequence Editor, expose .NET assemblies, TestStand does not support the use of undocumented .NET APIs.

Related tasks:

- [Installing TestStand](#)

Related information:

- [Communicating with an SCM Server in TestStand](#)

Updates and Changes for TestStand Extended Support Versions

Browse updates and changes made in TestStand versions on extended support.



Note If you cannot find changes for your version, it might be a more recent version, documented as a new feature. Or, your version might not have included user-facing updates. You can find more information about non-visible changes, such as bug fixes, compatibility updates, and stability adjustments or maintenance adjustments, in the product **Release Notes**, available on ni.com.

TestStand 2022 Q4 Changes

Learn about new features, behavior changes, and other updates in TestStand 2022 Q4.

Python Adapter Improvements

- **Namespace Support** — The Python Adapter now supports using namespaces when loading Python modules. Previously, TestStand would use only the file name when loading a Python module, which would create a conflict when loading modules in different directories with the same name. You can now specify namespaces for Python modules and use modules with the same file name.
- **Debugging Enhancements** — You can now configure the Python Adapter to allow updates to Python code modules during execution. If you enable this option prior to execution, TestStand will reload an updated Python module without unloading other modules or restarting TestStand execution.
- **Improved Information Displayed** — If you install the Jedi package, TestStand can now display additional Python elements used in your code modules. With Jedi, you can configure the step settings for a Python step to use inner classes, methods and attributes from base classes for a selected child class, and classes, attributes, and functions available in the current namespace.

Expression Editor Improvements

- **Delimiter Highlighting**—The expression editor adds new visual assistance for finding and manipulating nested delimiters such as parentheses, brackets, and braces. Place your cursor to the left of a delimiter and the expression editor makes the matching pair bold. Click <Ctrl +]> to toggle the cursor between the matching delimiters, or <Ctrl + Shift +]> to select the matching delimiters and everything between them.
- **Fixed Width Font**—On machines where Lucida Sans Typewriter is installed, the expression editor uses that font to display text in the expression editor. When the font is absent, the expression editor uses the default TestStand font. One method for obtaining Lucida Sans Typewriter is to install Microsoft Office applications on the machine.

Floating-Point Number Representation in XML Sequence Files

TestStand 2022 Q4 fixes a bug in how floating-point numbers were stored in XML sequence files. Sequence files saved in binary format are unaffected by the bug. The fix for this issue is backwards compatible with existing sequence files but may produce unexpected differences when saving sequence files created with older versions of TestStand.

- **Bug Details**—TestStand Number objects with the default representation are standard IEEE double-precision floating point data types, sometimes called `float64`. This type corresponds to the double data type in languages such as Java and .NET. In most cases C/C++ doubles use this format as well.

`Float64` is a binary data format. When TestStand stores Number objects in XML sequence files, the binary `float64` format is converted to a base-10 (decimal) format. The conversion from binary to decimal format is not exact in all cases. If there is insufficient precision, the round-trip conversion may lead to a different value when the file is read. For IEEE double-precision floating-point numbers, 17 decimal digits are required to guarantee stable round-trip conversion. Previous versions of TestStand stored 16 decimal digits, causing the bug.



Note Binary sequence files are not affected by this issue because binary sequence files store floating point numbers in their raw binary format.

- Bug Fix Details—TestStand 2022 Q4 uses 17 decimal digits when storing TestStand Number objects (with the default float64 representation) in XML sequence files.
- Impact—This change impacts saving sequence files. Existing sequence files will continue to load as before. (There was no issue with the loading code. The issue was purely the missing digit in the saved file.) Just as before, the value of number of objects read from an XML sequence file saved using prior versions of TestStand may differ from the value originally saved.

The value read will match the value read with older versions of TestStand, so this should not cause compatibility issues. However, if you modify an XML sequence file created with an older version of TestStand, save the file with a different name, and then compare it to the original version using a differ, you may notice very small numeric differences in the values of saved numbers. For example, in some cases the original file may contain "3122.55" while the 2022 Q4 (22.0) version of the file contains "3122.5500000000002". This change is a result of the fact that "3122.5500000000002" is actually a more accurate numeric encoding of the underlying binary format in this example.



Note The difference in numeric value will be small even though the length of the decimal encoding may be longer.

Once a sequence file has been saved using TestStand 2022 Q4, round-trip conversions between binary and XML format will be stable.

New API and UI Controls

TestStand 2022 Q4 includes the following new enumeration and constant values, properties and methods.



Note This list may not be complete.

- Enumeration and Constant Values
 - `UIStyleColor_Background`
 - `UIStyleColor_Background2`
 - `UIStyleColor_Foreground`
 - `UIStyleColor_GridLine`
 - `UIStyleColor_HeaderForeground`

- `UIStyleColor_SelectedBackground`
- `UIStyleColor_SelectedForeground`
- `UIStyleFontType_Proportional`
- `UIStyleFontType_Monospaced`
- `UIStyleFontType_SectionHeader`
- **Properties**
 - `GetColor`
 - `GetFontTypeName`
 - `GetFontSize`
- **Methods**
 - `ReloadModifiedModulesDuringExecution`



Note Although some TestStand components, such as the TestStand Sequence Editor, expose .NET assemblies, TestStand does not support the use of undocumented .NET APIs.

TestStand 2021 SP1 Changes

Learn about new features, behavior changes, and other updates in TestStand 2021 SP1.

Enhanced Python Module Debugging

- The Python Adapter now supports step-into debugging of Python modules. Users who want to perform step-into operations with Python modules must install additional software, including Visual Studio Code.

Support for HTML Reports in Modern Browsers

- TestStand 2021 SP1 removes the dependency on Microsoft Internet Explorer for viewing TestStand reports. You can now view TestStand default reports in a modern browser without manually changing security settings.

Improved Error Reporting

- When possible, TestStand now provides information indicating where in the

sequence file the error occurred.

- The error dialog box now displays information regarding the cause and possible resolutions for the error.

Updates to Input and Output Stream Parameter Capabilities

- Capabilities added to import parameters from and export parameters to CSV files.
- The Test Vector table is now marked as out-of-sync when changes are made to dynamic parameters in the Step Settings.

New and Obsolete API and UI Controls

TestStand 2021 SP1 includes the following new enumeration and constant values, properties and methods.



Note This list may not be complete.

- Enumeration and Constant Values
 - `CommonDlgOptions_DisableGotoLocation`
 - `ReportViewButton_OpenReportLocation`
 - `VersionConstant_PatchVersion`
- Properties
 - `PatchVersion`
 - `SectionHeaderFontName`
 - `SectionHeaderFontSize`
- Methods
 - `DebugJustMyCode`
 - `DisplayErrorDialog`
 - `EnableDebugging`
 - `PythonExecutablePath`
- Obsolete Properties, Methods, and Events
 - `ReportViewButton_FontSize`



Note Although some TestStand components, such as the TestStand Sequence Editor, expose .NET assemblies, TestStand does not support the use of undocumented .NET APIs.

Related tasks:

- [Debugging Python Modules](#)

TestStand 2021 Changes

Learn about new features, behavior changes, and other updates in TestStand 2021.

Sweep Loop Step Enhancements

TestStand 2021 expands the functionality of the existing Sweep Loop step. These updates include the following changes:

- You can create nested and parallel sweeps of parameters by changing the level of one or more parameters in a Sweep Loop step.
- TestStand automatically creates a variable when you specify a parameter name.
- You can choose between Static and Dynamic mode to specify whether values for a Strategy are specified using constants or expressions.
- You can use the Table View to interact with values and test vectors resulting from a sweep across selected parameters at edit time and run time. This functionality includes the ability to enable or disable, filter, and export test vectors.
- At execution time, use Table View to monitor captured data as well as the parameters in the Sweep Loop step.
- Added functionality to import and export parameter and test vector data to or from a CSV file.

Dropped Support for LabVIEW NXG Features

The following LabVIEW NXG features in TestStand are no longer supported as of TestStand 2021:

- LabVIEW NXG adapter
- LabVIEW NXG operator interfaces

New API and UI Controls

TestStand 2021 includes the following new enumeration and constant values, properties and methods.



Note This list may not be complete.

- Enumeration and Constant Values
 - `AutoCreateVariableLocationOption_Locals`
 - `AutoCreateVariableLocationOption_Parameters`
 - `AutoCreateVariableLocationOption_FileGlobals`
 - `AutoCreateVariableLocationOption_StationGlobals`
 - `EdgeStyle_UI`
- Properties
 - `AutoCreateVariableLocation`
 - `FrameEdgeUIStyleColor`
 - `GridLineColor`
 - `HeaderForegroundColor`
 - `InRedo`
 - `InUndo`



Note Although some TestStand components, such as the TestStand Sequence Editor, expose .NET assemblies, TestStand does not support the use of undocumented .NET APIs.

TestStand 2020 Changes

Learn about new features, behavior changes, and other updates in TestStand 2020.

TestStand Sequence Editor Visual Design Updates

TestStand 2020 introduces a streamlined visual design to the TestStand Sequence Editor. In addition to cosmetic changes, the sequence editor has the following changes:

- By default, the sequence editor toolbar now only displays buttons that are global in scope. Buttons that provide functionality restricted to a single window or pane are now located in the relevant window or pane.
- Panes or tabs that provide configuration options for a single window are fixed to the relevant window.

- On the Steps pane, steps now have syntax highlighting. The icon color for a step indicates which type of step it is.
- Checkboxes in the Select Sequence File Callbacks dialog box simplify the process of selecting callback sequences.

Python Adapter Improvements

TestStand 2020 adds support for Python 3.8. You can now use the Python Adapter to perform the following tasks:

- Pass COM objects between TestStand and Python code modules.
- Map enums in TestStand to enums in Python code modules.
- Map arrays of numbers in TestStand to NumPy arrays in Python code modules.
- Include or exclude subproperties of a container of a named data type when passing it between TestStand and a Python code module.
- Enter either an individual file or a directory as a code module path for the Python Adapter.

Regular Expression Support

Regular expressions are now a supported comparison type for String Value Test steps. You can also use regular expressions as parameters for the `CheckStrLimit`, `FindPattern`, `MatchPattern`, and `SearchPatternAndReplace` expression functions.

InstrumentStudio Integration Improvements

TestStand 2020 introduces improvements to integration with InstrumentStudio. You can now perform the following tasks:

- Use Python code modules with IO Configuration step types.
- Pass the `NI_IOSession` data type between TestStand and Python code modules.

TestStand 2019 Changes

Learn about new features, behavior changes, and other updates in TestStand 2019.

Quick Drop Dialog Box

In previous versions of TestStand, you could only insert objects into the TestStand Sequence Editor using mouse clicks. TestStand 2019 introduces the Quick Drop dialog box, an easy way of inserting steps, sequences, and variables directly from a dialog without using mouse clicks.

Marking Sequence Analyzer Messages as Ignored in Sequence Files

The sequence analyzer provides the ability to mark analysis messages as ignored in sequence files. Marking an analysis message as ignored in a sequence file stores the message in the sequence file so that the sequence analyzer does not report the message for subsequent analysis of the sequence file. When you mark a message as ignored, the sequence analyzer now prompts for text that serves as a justification to ignore the message. The sequence analyzer stores the justification text with the analysis message and displays it in the Analysis Results pane when displaying ignored messages.

TestStand Deployment Utility Updates

The TestStand Deployment Utility added the following features:

- Support for building packages and offline package installers in the 32-bit TestStand Deployment Utility.
- A fourth field to the deployment version number and package version number in package distributions, using the format X.X.X.X. You can auto-increment the fourth field of the package version after each successful build, allowing you to rebuild packages without changing the major.minor.patch fields of the package version. For MSI Based Installers, the fourth field is not present to maintain compatibility with previous versions of TestStand, and versions continue to use the format X.X.X.
- Filtering options in the Package Dependencies dialog which allow you to configure the relationship of dependent packages. These options allow you to filter by the visibility level, package type, or specific search text.
- Support for deploying steps which use the Python adapter. The deployment utility includes any Python files included. TSDU also provides information for what versions of the Python interpreter are required, and what virtual environments are used by steps in the deployment.
- An option to configure whether code modules of step types are deployed for

LabVIEW and LabVIEW NXG. When enabled, TSDU includes code modules in substeps of custom step types you create.

- Support for deploying files specified in the Override Module settings for LabVIEW steps.

Multicore Scaling Improvements

TestStand 2019 includes extensive optimizations for multicore processors. These improvements can significantly increase throughput for systems with high core counts running short duration tests on multiple UUTs in parallel. Performance improvements depend heavily on the specifications of your system.

PDF Report Generation

TestStand 2019 introduces an option to generate PDF reports which look the same as the report generated through the result processor. The PDF report works for all formats except text. Even if OTF is enabled, PDF reports are generated only at the end of execution (PDF reports will not be generated for intermediate reports).

Navigate to **Configure»Result Processing**, click the icon in the Options column to launch the Report Options dialog, then select **Generate PDF Report**.

Data Streams Enhancements

TestStand 2019 adds improved support for the data streams features.

- **Stream Loop Step Type**—Define a block of steps that execute for each element in an `InputRecordStream`. Optionally, you can specify an `OutputRecordStream` to write a record to at the end of each iteration.
- **CSV Record Prototypes**—CSV record streams now support specification of field names and data types. Use record prototypes to automatically map columns in a CSV file to records in a field.

Parametric Sweep

TestStand 2019 adds the Sweep Loop step type to support Parametric Sweep applications. Use the Sweep Loop step type to define a set of sweep parameters and a block of steps to execute for each set of values the parameters sweep over. You can

specify the range of each sweep parameter using one of several strategies. The default strategy is Start/Stop/Step. Other strategies include defining the sweep parameter as a constant value or specifying each value in the range using an array. Optionally, you can specify an `OutputRecordStream` to write sweep parameters (including outputs) at the end of each iteration.

Switching between Packed Project Libraries and VIs

The LabVIEW adapter now allows you to switch between packed project libraries and source VIs without having to reconfigure the step. This makes it easier for you to debug your packed project libraries. In previous versions of TestStand, you would have to reconfigure the step to point to the source VI in order to debug the issues present in a packed project library being called from a TestStand step. Now you can specify the packed project library source in the same step, which allows for seamless debugging. Additionally you can rebuild the packed project library from TestStand without switching to LabVIEW. In order to use this feature, navigate to the Override Module Settings window from the LabVIEW Step Settings panel to specify the source files (Project, VI, Build specification) of the packed project library. This window also provides a per-step override option, which allows you to choose between source VI or packed project library.

In the LabVIEW Adapter Configuration dialog box, you can override the module settings for all LabVIEW code modules to run either a source VI or a packed project library. You can also select the option of automatically building the packed project library. View these options on the Advanced tab in the LabVIEW Adapter Configuration dialog box.

LabWindows/CVI Integration Improvements

TestStand 2019 includes several improvements to the LabWindows/CVI adapter for TestStand that make the programming and debugging experience easier. There are three requirements for using these features:

- Install LabWindows/CVI 2019.
- In the LabWindows/CVI 2019 ADE, enable **Add NI Type Information resource to DLL** under **Target Settings»Type Information**.
- Rebuild DLLs that TestStand steps call using the LabWindows/CVI adapter.

Improved Handling of C Structs

Older versions of TestStand required users to manually create objects to represent C structs. This process was complicated, prone to error, and it required users to manually update their TestStand object every time the struct changed.

LabWindows/CVI 2019 embeds information about structs into the LabWindows/CVI DLL and TestStand 2019 can read it. The TestStand LabWindows/CVI adapter now has a **Create/Update Data Type** button that can automatically generate a TestStand custom type definition to represent the C struct. If the struct changes in the LabWindows/CVI code module, users can update the TestStand type with a click of a button.

Improved Handling of Enums

Older versions of TestStand required the user to manually create enums to match the enums used in C. Enum information is now embedded into the LabWindows/CVI DLL, so users can automatically import and update their TestStand enums. When a function with an unknown enum is selected, the **Create/Update Data Type** option will appear.

Source Code Navigation

Older versions of TestStand required users to locate the `.c` file and the LabWindows/CVI project the first time they attempted to step into a LabWindows/CVI code module. LabWindows/CVI 2019 embeds the location of the `.c` and project file into the LabWindows/CVI DLL. When TestStand loads the DLL, it automatically fills this information into the Specify Module panel.

Pointer vs. Array Ambiguity

In C, pointer and one-dimensional array parameters have the same data type in function signatures. Even if you specify an array parameter using square braces, it is still passed as a pointer. Because C treats both syntaxes identically, earlier versions of TestStand required users to manually select whether a parameter was an array or pointer every time they called a LabWindows/CVI module with them.

TestStand 2019 automatically detects if a parameter is declared as an array using square brace syntax, and automatically defines the parameter as an array in the adapter. If the parameter includes an array size, Test Stand will automatically use that

as the array size.

Additionally, users can use the `///OUT` syntax to specify a parameter as a pointer. This will allow users to avoid the prompt. For more information, see the following tutorial: [Using Source Code Tags to Enhance LabWindows™/CVI™ Code Documentation](#).

Python Adapter

The Python Adapter provides advanced functionality for calling Python code modules from TestStand. You can use the Python Adapter to complete the following tasks:

- Execute Python scripts (`.py`) on disk by:
 - Calling a function defined in a module.
 - Getting/Setting the attributes defined in a module.
 - Creating a class instance.
 - Calling a member function or static function defined in the class.
 - Getting/Setting member attributes or static attributes defined in the class.
- Execute Python scripts in Python versions 2.7 or 3.6+.
- Execute Python scripts out-of-process in the CPython interpreter.
- Convert data between Python and TestStand variables.
- Store/reuse the Python object in a TestStand variable (Object Reference).

You can also use multiple Python interpreter sessions to perform the following tasks:

- Execute Python scripts in parallel.
- Use multiple Python versions, such as 2.7 and 3.6, simultaneously in TestStand.

When you specify a module for a step, the TestStand Sequence Editor displays the Python Module tab. TestStand User Interfaces launch the Edit Python Call dialog box.

Use the Python Adapter Configuration dialog box to configure the Python version and Python virtual environment path to use with the Python Adapter.

Before using the Python Adapter in TestStand, you must install the required version of the CPython interpreter.



Note During installation of the CPython interpreter, you must enable the

option for adding the Python path to the environment variables.

IO Configuration Step Types

TestStand 2019 adds IO Configuration step types to control NI Modular Instruments NI-SCOPE, NI-DCPower, NI-DMM and NI-FGEN using InstrumentStudio. You can initialize an IO session, apply an IO configuration to an existing IO session and close an existing IO session for NI Modular Instruments. You can also use the Sweep Loop step type to sweep on the instrument attributes specified by an IO session.



Note To use IO Configuration step types, you must download and install InstrumentStudio and iPXI driver software.

Deprecation of the INI File Format

Configuring TestStand files to save using the INI file format is deprecated in TestStand 2019. Saving files in INI file format will be deprecated in a future release of TestStand. In TestStand 2019, the following user interfaces now restrict you to selecting only the XML or binary formats:

- The File Format Options dialog box displayed when you select the File tab of the Station Options dialog box.
- The General tab of the Sequence File Properties dialog box displayed when you select **Edit » Sequence File Properties** in the Sequence Editor.
- The Update Sequence Files tool displayed when you select the Tools menu in the Sequence Editor.



Note If you change a file you stored in an SCC system from INI (or XML) to binary, you must update the SCC system to set the file type as binary. Otherwise, the SCC provider might incorrectly process the binary content of the file as white space and produce a corrupt file.

Sequence File API

TestStand 2019 introduces enforcing absolute paths for input parameter for the TestStand API's `SequenceFile.Save()` and

`SequenceAdapter.GetSequenceFile()` methods. Previously, these methods did not enforce absolute paths for the input parameters.

Related information:

- [Documentation Tags for Source Code in LabWindows/CVI](#)
- [Python Homepage](#)

TestStand 2017 SP1 Changes

Learn about new features, behavior changes, and other updates in TestStand 2017 SP1.

LabVIEW Run-Time Engine Error Reporting Improvements

The LabVIEW Adapter can provide detailed error information about broken VIs when used with the LabVIEW 2018 or later run-time engine. Refer to Error Reporting for Broken VIs in the LabVIEW Run-Time Engine for more information.

Use of TestStand Tools Without Activation

TestStand 2017 SP1 enables the use of several TestStand Tools without activation.

User-Editable Installer Executable Names

You can now set the installer name when you create distributions using the TestStand Deployment Utility.

Property Loader Comment Support for CSV, Excel, and Text Files

You can now use `//` characters to denote a comment in a CSV, Excel, or text file that the Property Loader step ignores when reading.



Note When a TestStand 2017 SP1 sequence file includes steps that contain comments and the file is opened in TestStand 2017, the comments cannot be ignored and may result in an error reading the file at runtime.

INI File Support

Saving to and reading from TestStand files that use the INI file format will be deprecated in future versions of TestStand.

You can change the file format of a TestStand file to XML or binary formats in the following ways:

- Use the File Format Options dialog box, by selecting the **File** tab of the Station Options dialog box.
- Use the **General** tab of the Sequence File Properties dialog box, by selecting **Edit»Sequence File Properties** in the Sequence Editor.
- Use the Update Sequence Files tool, by selecting the **Tools** menu in the Sequence Editor.



Notice If you change a file you store in an SCC system from XML or INI to binary, you must update the SCC system to set the file type as binary. Otherwise, the SCC provider might incorrectly process the binary content of the file that appears to be white space, resulting in a corrupt file.

Change in Installer File Name

(Windows) When you build an installer in TestStand, the default name of the executable changed from `setup.exe` to `install.exe`. This behavior change may have potential impact to post-build processes that rely on the `setup.exe` name. To specify a different name for the executable, use the Installer name option on the Product Information page of the Installer Properties dialog box.

Related concepts:

- [Activating and Licensing TestStand](#)

TestStand 2017 Changes

Learn about new features, behavior changes, and other updates in TestStand 2017.

Data Streams Step Types

TestStand Data Streams provide a simple way to read data from or write data to CSV (comma separated value) files. Basic Data Streams functionality is provided by the Data Streams step types as well as an update to the existing For Each step type. Use these step types to quickly and easily create sequences that consume or produce CSV file data.

Data Streams step types are built on an underlying public API that you can call directly to handle more complex applications. At the core of the Data Streams feature are two COM (Component Object Model) interfaces: `InputRecordStream` and `OutputRecordStream`. Advanced users can create custom implementations of these interfaces to enable TestStand sequences to work with any producer or consumer of data that has a logical structure similar to a table.

To get started using the feature, use the example code provided with TestStand 2017 at `<TestStand Public Directory>\Examples\Fundamentals\Using Data Streams`.

Crash Recovery for TestStand Applications

TestStand 2017 has the option to generate a log containing execution information and use the NI Error Reporting Service, NIER, to automatically generate a crash dump file. You can use a crash dump file to diagnose the root cause of a crash. In cases where the crash is in an NI product, you can send the crash dump file to NI support engineers for diagnosis. TestStand also offers a way to register a custom sequence to execute in an external process in response to a crash.

TestStand Crash Log—When a crash occurs TestStand generates a log file in the directory `<TestStand_LocalAppData>\CrashLogs`. Only the last 100 crash logs are saved, after which the earliest log file will be deleted and a new log file is created.

NIER Crash Dump—Using the same error reporting service as LabVIEW, TestStand generates a NIER crash dump in `<TestStand_LocalAppData>\NIERdump`.

TestStand Crash Log and Dump File

You can use a configuration file to specify settings for the dump and TestStand specific log file. The configuration file is an `.ini` file in the same directory as the TestStand application. The name of the configuration file must be same as the executable name of the TestStand application.

In the configuration file, create a section with the same name as the name of the executable. Use the following tokens and values to specify the options:

Token Name	Value	Purpose
TSLogType	None	Do not create TestStand crash log. By default, TestStand crash log is created.
NIER	False	Do not create NIER crash dump. By default, NIER crash dump is created.
NIERDumpType	Full	Create full (instead of mini) NIER crash dump. By default, mini dump file is created.
LogVariables	True	Enable logging of Locals, Parameters, and FileGlobals in the TestStand specific log file. By default, Locals, Parameters, and FileGlobals are not logged in the TestStand log file.



Note The crash log and dump file are not sent to any server. The crash log and dump file are not generated if the application crashes before creating an instance of the TestStand Engine.

Code modules should not register to functions like

`SetUnhandledExceptionFilter`, `set_invalid_parameter_handler`, and `set_purecall_handler` to handle application crashes. Crash logging due to running out of memory is not supported.

Registering a Crash Recovery Sequence

You can register sequences to respond to a crash in your TestStand application to return any hardware or other system assets to a good state. The sequences are executed by an external process and run without a process model or a logged-in user. Since multiple crash recovery sequences can be registered, the order in which the

sequences will be executed in the order in which they are registered.

Any errors encountered in executing crash recovery sequences are recorded in a log file, and a dialog displays the location of the log file. Similar to the main application TestStand crash log, this log file stores the errors from a maximum of 100 crash callbacks.

To register a sequence for execution in the case of a crash you must call `Engine.RegisterSequenceToExecuteOnCrash` with the sequence name and file path to be executed. This function returns a unique ID that can be used to unregister the sequence later with a call to

`Engine.UnregisterSequenceToExecuteOnCrash`.

Support for Package Distribution in the TestStand Deployment Utility

TestStand Deployment Utility now supports building package distributions in addition to MSI-based installers. Use the **Output Type** control on the **Mode** tab to configure the TestStand Deployment Utility to build a package-based installer. This control has 3 settings:

- **Deployable Image Only**—The deployment files are moved to the specified image directory on the System Source tab.
- **MSI-Based Installer**—In addition to the deployable image, the deployment utility creates an installer to distribute the system. When you select this option, the Installer tab is enabled, which you can use to configure the installer.
- **Package-Based Distribution**—In addition to the deployable image, the deployment utility creates a package distribution. When you select this option, the Package Distribution Options tab is enabled, which you can use to configure the package distribution. You can configure the package output type using the options below:
 - **Single Package**—The files in the image directory are packaged into a single `.nipkg` file. Dependencies you select are referenced by the package, and must be installed on the deployment target to install the package.
 - **Repository**—In addition to the main package, all dependent packages are also included in the deployment (the repository). A feed is generated which references all included packages in the repository. To install the deployment, you can register the feed in NI Package Manager on the target machine.
 - **Package Installer**—In addition to the main package, all dependent packages

are also included in the deployment. A standalone installer is generated which installs all the packages on a machine. The installer will also install NI Package Manager if it is not present on the machine.

Configuring the Package Settings in the TestStand Deployment Utility

Package distribution settings are configured on the Installer tab. The package name is generated from the Installation name you specify. To configure additional settings for the package, click the **Advanced Options** button.



Note The **Include Required Certificate Packages** option determines whether to include certificates packages, which reduce the number of prompts you receive when installing the package. This setting applies only to feeds and package installers.

To include package-based dependencies, click the **Dependencies** button. For each dependency you select, you can configure the dependency level:

- **Required**— The package cannot be installed unless the dependency is on the system or can be found by NI Package Manager through a registered feed. These packages are included in the deployment in feed or package distribution mode.
- **Recommended/Optional**— The package can be installed without these dependencies. They are not included in feed or package distribution mode.

The following features are currently not supported in package mode:

- Advanced File Options (creating shortcuts, program items, or registering as an ActiveX server)
- Custom Commands
- Including readme files or license agreements
- Media spanning

Support for LabVIEW NXG in the TestStand Deployment Utility

TestStand Deployment Utility supports deploying sequence files which call LabVIEW NXG code modules. LabVIEW NXG steps can be configured in the following ways:

- **Project and GLL are Both Specified**— These steps can execute using the LabVIEW NXG development system to execute the VI directly, or using the LabVIEW NXG Run-Time Engine to execute the VI within the GLL. In this case, TestStand enforces that the specified GLL must match the GLL output configured in the project and component in LabVIEW NXG. When deploying these types of steps, TestStand Deployment Utility automatically regenerates the GLL from the project before deploying to ensure that the latest code is used.
- **GLL Specified, Project Not Specified**— These steps must be executed using the LabVIEW NXG Run-Time Engine to execute the VIs built into the GLL. When deploying these types of steps, the GLL file is deployed as-is because no source files are present to rebuild it.
- **Project Specified, GLL Not Specified**— These steps can execute using the LabVIEW NXG Development System to execute the VI directly. Deployment is not supported for these steps by TestStand Deployment Utility. An error is generated indicating that a GLL is required.



Note If a LabVIEW NXG file is included in a deployment but not referenced by the sequence file, TestStand Deployment Utility will not process the file and will include it as-is.

Configuring LabVIEW NXG Files in the TestStand Deployment Utility

After analyzing a sequence with LabVIEW NXG code modules, the referenced GLLs are displayed in the distributed files tab. You can configure the destination for the GLL file. Like other code module types, TestStand Deployment Utility ensures that the sequence file is updated to reference the correct files if the files are deployed within the same base destination as the calling sequence file.

To include LabVIEW NXG source files in the deployment, click **LabVIEW options**, and select the LabVIEW NXG options tab. Enable the **Include LabVIEW NXG Source Files** option. With this option selected, LabVIEW NXG projects, components, and VIs will also be displayed in the distributed files tab. Only referenced files are displayed in the distributed files tab, but all files included in the project will be included in the deployment. You can configure the destination for the source project, but not for contained components and VIs; these files will be deployed to a location relative to the project.

Deploying the LabVIEW NXG Run-Time Engine in an MSI-based Installer in the TestStand Deployment Utility

Because the LabVIEW NXG Run-Time Engine is a National Instruments package it is not directly available as a dependency in an MSI-based installer. However, if you enable the **Automatically Include Required Installers** option in the Drivers and Components dialog, TestStand Deployment Utility will include the Run-Time Engine package within the installer folder. When the installer is executed, it will invoke NI Package Manager to install the Run-Time Engine package.

Limitations in the TestStand Deployment Utility

The following LabVIEW step features are not currently supported with LabVIEW NXG Steps:

- Generating a PPL for deployment. This feature is no longer necessary because LabVIEW NXG does not create PPLs.
- Merging of LabVIEW NXG projects.

Early Access Support for LabVIEW NXG

TestStand 2017 provides early access support for LabVIEW NXG. The LabVIEW NXG Adapter is suitable for developing simple test systems, or augmenting existing test systems, but has limitations that may not be acceptable for all applications. National Instruments recommends that you fully evaluate LabVIEW NXG before using it in a TestStand system.

Consider the following topics when evaluating LabVIEW NXG for your test system:

- Processor architecture—The LabVIEW NXG Adapter is only available in 64-bit TestStand.
- LabVIEW NXG Classes—TestStand 2017 does not support LabVIEW NXG classes.
- GLL Support—TestStand 2017 enables the ability to build GLLs for use with the run-time engine. GLLs can be executed through the LabVIEW NXG adapter but cannot be called directly from VIs.
- Debugging—LabVIEW NXG 2.0 does not support building debuggable GLLs. TestStand 2017 supports debugging LabVIEW NXG code modules in the development environment.

- COM Automation Support—Invoke nodes are not available in LabVIEW NXG. All Properties and Methods of the TestStand API are available through the TestStand palette in LabVIEW NXG.
- Modules and Toolkits—LabVIEW NXG is not supported in TestStand Semiconductor Module applications.
- Data types Support—The following data types present in LabVIEW NXG are not supported in the LabVIEW NXG Adapter:
 - Variants
 - Digital Waveform
 - Digital Table

LabVIEW NXG Adapter

TestStand 2017 offers early access support for calling into code modules developed in LabVIEW NXG 2.0 with the LabVIEW NXG Adapter. The LabVIEW NXG Adapter provides advanced functionality for calling VIs from TestStand. You can use the LabVIEW NXG Adapter to complete the following tasks:

- Call VIs that exist in a LabVIEW NXG project or a LabVIEW NXG GLL.
- View the VI description and parameter descriptions through the Module tab in Step Settings.
- Run VIs using the LabVIEW NXG Development System.
- Run VIs in GLLs using the LabVIEW NXG Run-Time Engine.
- Create and edit VIs from TestStand if the LabVIEW NXG Development System is installed.
- Auto-build a LabVIEW NXG GLL from the LabVIEW NXG project when executing in the LabVIEW NXG Run-Time Engine.

For a LabVIEW NXG test step, specify the module in the Module tab of the Step Settings pane by defining the Project or GLL path, the VI name, and the desired parameters. The LabVIEW NXG Adapter requires the Project or GLL Path and the VI name to be defined.

The LabVIEW NXG Adapter can run VIs using the LabVIEW NXG Development System or the LabVIEW NXG Run-Time Engine. You must build your VIs into a GLL containing your VIs when using the Run-Time Engine. Additionally, recompilation of GLLs is required when modifications are made to the original VIs.

Use the **LabVIEW NXG Adapter Configuration** dialog box to switch between executions

using the LabVIEW NXG Development System and the LabVIEW NXG Run-Time Engine.

The Adapter Configuration also includes the option to **Automatically Build Missing GLLs at Start of Execution**. This option allows you to run a step in the Run-Time Engine which only has the project and VI specified. The LabVIEW NXG Adapter builds the GLL if necessary.

Use the **LabVIEW NXG Advanced Settings** dialog box in the Step Settings pane to access the option **Always Run VI in LabVIEW NXG Run-Time Engine**. This option allows you to execute an individual code module in the Run-Time Engine, regardless of the server setting for the LabVIEW NXG Adapter.

TestStand LabVIEW NXG Conversion Utility

Use the TestStand LabVIEW NXG Conversion Utility to convert TestStand files that use LabVIEW code modules to refer to their converted LabVIEW NXG code modules. You can convert TestStand Workspaces (`.tsw`), Sequence Files (`.seq`), TestStand Deployment Specification files (`.tsd`), and the folders containing those files using the TestStand LabVIEW NXG Conversion Utility.



Note VIs are only converted by the Conversion Utility if they are referenced by any of the above-mentioned file types. You cannot convert standalone VIs using this tool.

The Conversion Utility application window contains the following panes:

- **Files pane** — This pane displays items added for conversion, their dependencies, and their status. The pane also indicates duplicate files and files that are called in a recursive loop (a file that calls itself at some point in its dependency hierarchy) that were added for conversion. You can navigate to the file from this pane by right clicking the file and selecting the **Open Original** option.
- **Preview pane** — This pane displays a preview of the output directory. When you start a preview, preconversion checks run on the files added for conversion, and any errors or warnings are reported in the log.
- **Log pane** — This pane displays and logs error, warning, and information messages that are generated during the conversion process.
- **Call Hierarchy pane** — This pane displays the call hierarchy of items and is shown

when you click the **Call Hierarchy** button on any log message. The **Call Hierarchy** button appears only if the log message is about an item undergoing conversion.

TestStand API in LabVIEW NXG

TestStand 2017 includes full access to the TestStand API in LabVIEW NXG through the addition of a TestStand palette to the Functions palette. The palette includes VIs to call into the TestStand API as well as basic COM automation functionality to handle references to TestStand classes.



Note LabVIEW NXG does not have an Automation front panel control for use with subVIs in the controls palette. If you need to develop a subVI with an Automation control for input, you can place an Automation control on the block diagram.

Operator Interfaces in LabVIEW NXG

TestStand 2017 includes early access support for developing TestStand operator interfaces in LabVIEW NXG. TestStand 2017 populates the LabVIEW NXG controls palette with the same user interface controls as well as the same ApplicationMgr, ExecutionView, and SequenceFileView controls as is offered in other development environments. Overall development of TestStand operator interfaces offers a similar experience in LabVIEW NXG as LabVIEW.

To help get started with operator interface development in LabVIEW NXG, new versions of the Simple and Full-Featured UIs written fully in LabVIEW NXG are available as shipping examples with TestStand 2017. The examples can be found in the `<TestStand Public>\TestStand 2017\UserInterfaces\` directory.

HTBasic Adapter

In TestStand 2017 and later, the HTBasic Adapter is hidden by default. You can enable the HTBasic Adapter in 32-bit TestStand by selecting **Configure » Adapters** and opening the **HTBasic Adapter Configuration** Dialog Box.

INI Format

Saving to INI file format will be disallowed in future versions of TestStand.

Year-Based and Major.Minor Version Equivalents

Versions of TestStand that follow year-based versioning continue to use major.minor notation for some features. The following table lists year-based TestStand versions and the equivalent major.minor notation.

Table 3. TestStand Versions and the Equivalent Major.Minor Notation

Year-Based Version	Major.Minor Internal Version	Engine Version
TestStand 2025 Q2	24.9	23.0.0
TestStand 2024 Q4	24.8	23.0.0
TestStand 2023 Q4	23.8	23.0.0.49152
TestStand 2022 Q4	22.0	22.0.0.49152
TestStand 2021 SP1	21.1	21.1.0.49154
TestStand 2021	21.0	21.0.0.49156
TestStand 2020	20.0	20.0.0.49180
TestStand 2019	19.0	19.0.0.170
TestStand 2017 SP1	17.1	17.1.0.130
TestStand 2017	17.0	17.0.0.184
TestStand 2016 SP1	16.1	16.1.0.52
TestStand 2016	16.0	16.0.0.193

Updates to Default Database Schemas

When the default database schema is updated in new versions of TestStand, the old default database schemas remain available as legacy schemas.

The following table summarizes changes to the default database schema:

Table 4. Changes to the Default Database Schema

Version	Default Database Schema Changes
TestStand 2020	The data types for the following columns in the PROP_NUMERICLIMIT table were changed from Numeric to String to enable TestStand to log INF, -INF, NAN and IND values when you use them as limits of Numeric Limit Test and Multi Numeric Limit Test steps. • LOW_LIMIT • HIGH_LIMIT • NOMINAL_VALUE • LOWER_THRESHOLD • HIGHER_THRESHOLD
TestStand 2016	The following columns were added to the PROP_NUMERICLIMIT table to log limit values when you use the EQT comparison type in Numeric Limit Test and Multi Numeric Limit Test. • THRESHOLD_TYPE • NOMINAL_VALUE • LOWER_THRESHOLD • HIGHER_THRESHOLD
TestStand 2012	The following columns were added to the UUT_RESULT table to log the part number of the UUT and data from the TestStand Offline Results Processor. • PART_NUMBER • TSR_FILE_NAME • TSR_FILE_ID • TSR_FILE_CLOSED
TestStand 4.1	The following changes were made to support logging additional results:

Version	Default Database Schema Changes
	- The tables listed below are no longer included and this information is logged in the PROP_RESULT table, which includes property result information and simple intrinsic property values. The PROP_RESULT table contains a foreign key that references the STEP_RESULT table. - STEP_PASSFAIL - STEP_CALLEXE - STEP_MSGPOPUP - STEP_PROPERTYLOADER - STEP_STRINGVALUE - MEAS_IVI_SINGLEPOINT - The following tables were added, each of which contains a foreign key that references the PROP_RESULT table: - PROP_BINARY—logs binary data, such as array values. - PROP_ANALOGWAVEFORM—logs analog waveform data, such as the LabVIEWAnalogWaveform, NI_IviWave, and NI_IviWavePair data types. This table replaces the MEAS_IVI_WAVE and MEAS_IVI_WAVEPAIR tables in previous versions of TestStand. - PROP_DIGITALWAVEFORM—logs digital waveform data, such as the LabVIEWDigitalWaveform data type. - PROP_NUMERICLIMIT—logs numeric limit data. This table replaces the MEAS_NUMERICLIMIT table in previous versions of TestStand. - The following new fields were added to the STEP_RESULT table: - STEP_ID—specifies the unique step ID. - CAUSED_SEQFAIL—specifies whether a step caused a sequence failure.

Related concepts:

- [Using Legacy Database Schemas](#)
- [PROP_NUMERICLIMIT Table Schema](#)
- [PROP_RESULT Table Schema](#)
- [PROP_BINARY Table Schema](#)
- [PROP_ANALOGWAVEFORM Table Schema](#)
- [PROP_DIGITALWAVEFORM Table Schema](#)
- [STEP_RESULT Table Schema](#)
- [Default TestStand Table Schemas](#)

Migrating to the Current Version of TestStand

If you are upgrading from TestStand 2010 or later, you can use the TestStand Migration Utility to migrate customizations and configuration settings from the earlier version of TestStand.

Consider the following when using the TestStand Migration Utility:

- The utility only moves files and does not update any other information or settings, such as registry keys.
 - If the `<TestStand Public>\Components` directory of the previous version of TestStand includes files installed by an installer other than the TestStand installer, you may need to use that specific installer to move the files. Files installed by a different installer migrated by the TestStand Migration Utility may not function correctly or uninstall properly from the `<TestStand Public>\Components` directory of your target version of TestStand.
 - The Migration Utility does not copy files in the `<TestStand>\Examples`, `<TestStand>\Tutorial`, or `<TestStand>\AdapterSupport` directories. These directories do not appear in the Files to Migrate column of the utility.
 - The TestStand 2024 Q4 Migration Utility does not migrate `<TestStand LocalAppData>\layout_current.bin`, `<TestStand LocalAppData>\SeqEdit.xml`, or `<TestStand LocalAppData>\UserInterface.xml` files from previous TestStand versions.
1. Launch the TestStand Migration Utility from the `<National Instruments>` folder in your start menu. If you have more than one version of TestStand installed on the computer, the utility launches a dialog box where you specify the previous version of TestStand you want to migrate from.



Note When you migrate from TestStand 2010 SP1 or TestStand 2010 to the current version of TestStand, the utility migrates the report and database options to the format for result processing configuration options for the current version of TestStand.

2. Select the version of TestStand you are migrating from. The utility checks your

computer for all the files you can migrate to determine if the files have changed since installation. The utility launches a status indicator dialog box during this process.

3. Use the **Files to Copy** control to select the files you want to copy. The utility selects files you added or modified in the version of TestStand you are migrating from by default.
4. Expand the directories in the **Files to Copy** control to view the files you can migrate and whether those files have changed since installation. The **Conflicts** column shows any conflicts that could affect the migration of your selected files.



Note The utility overwrites any file you create or modify in the current version of TestStand if the file also exists in the corresponding directory of the previous version of TestStand and if you select the file in the **Files to Copy** control. Although the utility creates a backup copy of all overwritten files, verify that the utility will not overwrite any files you intend to keep in the version of TestStand you are migrating to.

5. Click **Migrate Files** to begin the migration.
When the migration completes successfully, the utility generates a text file report that contains the results of the migration, including where the utility stored the file backups and which files were moved.

You can also upgrade manually, or migrate between 32-bit and 64-bit TestStand installations of the same version.

Related concepts:

- [64-bit TestStand and Migrating from 32-bit TestStand](#)

Related information:

- [TestStand Migration Guidance](#)

Migrating TestStand 2010 SP1 or Earlier Result Processing Options to the Current Version of TestStand

Process models from TestStand 2012 and later include significant changes to implement a plug-in architecture for model sequence files. As a result, the result

processing options files from TestStand 2010 SP1 or earlier are not compatible with TestStand 2012 or later process models without migration.

TestStand 2012 or later uses the plug-in mechanism to decouple result processing from the process models. As part of this change, the result processing options move from the `TestStandModelReportOptions.ini` and `TestStandDatabaseOptions.ini` files into a single `ResultProcessing.cfg` file located in the `<TestStand Application Data>\Cfg` directory, which stores configuration information for all result processing model plug-ins. Without migration through the TestStand Migration Utility or the Legacy Model Switcher tool to use legacy TestStand 2010 process models, results processing options files from TestStand 2010 SP1 or earlier will not function properly in TestStand 2012 or later.

The following table describes the migration actions you can select in the TestStand Migration Utility to migrate the result processing configuration options from TestStand 2010 SP1 or earlier to the current version of TestStand.



Note The TestStand Migration Utility does not support migrating the configuration files from TestStand 2010 SP1 or earlier if you modified the `ReportOptions` or `DatabaseOptions` data types defined in the default process models.

Migration Action	Description
Apply to new models	Creates a new configuration in a configuration set named <code>Migrated from TestStand <Version></code> . The utility automatically selects a unique name for the configuration if the utility has already used the default name in a previously completed migration. Refer to the migration report for the specific configuration names the utility uses.
Apply to legacy models	Copies the options files from TestStand 2010 SP1 or earlier to the current TestStand directory so you can use the previous configurations with the equivalent legacy TestStand 2010 process models in the version of TestStand you're migrating to. When you migrate database

Migration Action	Description
	options, the utility renames <code>TestStandDatabaseOptions.ini</code> to <code>TestStandDatabaseSchemas.ini</code> .
Apply to new and legacy models	Adds a new configuration in the Migrated from TestStand <Version> configuration set and copies the options files from the version of TestStand you're migrating from.

If you are upgrading from TestStand 4.2.1 or earlier, you can manually migrate result processing options for use with TestStand 2012 or later process models or the equivalent legacy TestStand 2010 process models.

Related information:

- [TestStand Migration Guidance](#)
- [Legacy Model Switcher](#)

Manually Migrating Result Processing Options from Any Version of TestStand

Complete the following steps to manually migrate result processing options from TestStand 2010 SP1 or earlier for use with the process models in the current version of TestStand.

1. Copy the `TestStandModelReportOptions.ini` and `TestStandDatabaseOptions.ini` files from the `Cfg` directory of TestStand 2010 SP1 or earlier to the `<TestStand Application Data>\Cfg` directory of the current version TestStand.
2. Rename `TestStandDatabaseOptions.ini` to `TestStandDatabaseSchemas.ini`.



Note This procedure is not supported for migrating TestStand 4.0 or earlier default database schemas. If you want to use TestStand 4.0 or earlier default database schemas in TestStand 2012 or later, click the **Reload NI Schemas** button on the Schemas tab of the Database Options

dialog box to launch the Reload NI Schemas dialog box, and click the **2.x to 4.0** button to reload the default database schemas from TestStand 4.0 or earlier.

3. Use the Legacy Model Switcher tool with the command-line argument `/migration` to migrate the report options and database options from legacy TestStand 2010 process models to the TestStand 2012 or later process models. Confirm that you have enabled the options to migrate report and database options from the legacy models.



Note The `/migration` command-line argument disables the type version checking the Legacy Model Switcher tool ordinarily performs. This automatically merges the types in the report and database options in the equivalent legacy TestStand 2010 process models with the types in the current TestStand process models. NI does not recommend migrating types that were not originally shipped with TestStand because doing so can lead to loss of data.

4. Launch the TestStand Sequence Editor.
5. Select **Configure»Result Processing** to launch the Result Processing dialog box.
6. Enable the **Show More Options** option.
7. Verify that the **Active Configuration** ring control specifies **Options from Legacy Models**.

To migrate result processing options to the equivalent legacy TestStand 2010 process models, complete only step 1. However, NI does not recommend using the equivalent legacy TestStand 2010 process models unless you encounter an incompatibility in an existing test system when you use the current TestStand process models or other circumstances prevent you from using the current TestStand process models.

Related concepts:

- [Process Model Changes in TestStand 2012 or Later and the Migration Utility](#)
- [TestStand Migration Utility](#)

Related information:

- [Command Line Arguments](#)

Fundamentals

Learn about TestStand concepts, including creating and debugging sequences, logging data, extending TestStand functionality, and integrating TestStand with other application development environments, such as LabVIEW and LabWindows/CVI.

If you are a new user, NI recommends that you complete the tutorials in ***Getting Started with TestStand*** to familiarize yourself with TestStand concepts and features.

Related concepts:

- [Database Logging](#)
- [Extending TestStand](#)
- [Effectively Using LabVIEW with TestStand](#)
- [Getting Started with TestStand](#)

Getting Started with TestStand

If you are using TestStand for the first time, NI recommends that you complete the tutorials in this manual. These tutorials begin with a general introduction to the TestStand Sequence Editor and continue by teaching you how to build sequences, manage test data, and customize TestStand functionality. Because the steps of the tutorials depend on elements from previous tutorials, NI recommends that you follow the chapter outline in order. Completed solution files are located in the <TestStand Public>\Tutorial\Solution directory.



Note The <TestStand Public> directory is located by default at C:\Users\Public\Documents\National Instruments\TestStand on Microsoft Windows 10/8.1/7.

1. [Introduction to TestStand](#)
2. [Loading and Running Sequences](#)
3. [Editing Steps in a Sequence](#)

4. [Debugging Sequences](#)
5. [Using Variables and Properties](#)
6. [Using Callback Sequences](#)
7. [Adding Users and Setting Privileges](#)
8. [Interactive Executions](#)
9. [Calling Sequences Dynamically](#)
10. [Customizing Reports](#)
11. [Calling LabVIEW VIs](#)
12. [Calling LabWindows/CVI Code Modules](#)

Introduction to TestStand

NI TestStand is a flexible and open test management framework for building, customizing, and deploying a full-featured test management system.

Creating a TestStand-Based System

Creating a TestStand-based test solution is a process that includes designing the test system, developing test sequences for units under test (UUTs), customizing the TestStand framework, debugging the test sequences, and deploying the test system to test station computers.

Design

Learn about the components of the TestStand framework, learn how to use those features, and understand when to customize that behavior. Plan the system architecture of the proposed solution. Determine the development environments to use to develop user interfaces and code modules in the solution. Consider how to manage the development and deployment of the solution, including how to organize files under development, how to deploy the files, and how to debug and update systems you deploy.

Refer to the section of this tutorial for more information about the components and features of TestStand and how to customize those features. Refer to the NI TestStand Advanced Architecture Series for more advanced concept and architecture information. This series is a suite of documents TestStand architects and developers created to provide more detailed information for experienced TestStand users with complex projects.



Note Before you start developing a test system, you must understand the concepts in the Deploying TestStand Systems portion of the NI TestStand Help so you can make appropriate design decisions that affect deployment as you complete development tasks.

Develop

Write test sequences for use on a test station computer. A test sequence is a series of steps that initialize instruments, perform complex tests, or change the flow of executions. Use the sequence editor or a custom sequence editor to write and edit test sequences.

Customize

Edit the default behavior of the TestStand framework depending on the needs of the application you create. You can customize reporting, database logging, process models, callback sequences, and user interfaces to create a unique, robust test solution for an application.

Debug

Ensure that the test sequences and any customized features execute correctly before you deploy the test system. Use the TestStand Sequence Analyzer in the sequence editor or the stand-alone sequence analyzer application during development or before deployment to find errors and enforce custom development guidelines you establish.

TestStand provides multiple features in the sequence editor or a custom sequence editor for debugging sequences, including tracing, breakpoints, conditional breakpoints, stepping through code, and watch expressions. The TestStand system development cycle is an iterative process, and you might have to debug an application multiple times.

Deploy

After you develop, customize, and debug a TestStand system, you can deploy the system to multiple test stations. The TestStand Deployment Utility simplifies the complex process of deploying a TestStand system by automating many of the steps involved, including collecting sequence files, code modules, configuration data for

instruments, and support files for the test system. You can also use the deployment utility to create an installer or patch distributions.

Major Software Components of TestStand

The major software components of TestStand include the TestStand Engine, sequence editor, user interfaces, module adapters, process models, and deployment utility.

TestStand Engine

The TestStand Engine is a set of DLLs that exports an ActiveX Automation server API. The sequence editor and TestStand User Interfaces use the TestStand API, which you can call from any programming environment that supports access to ActiveX servers, including code modules you write in LabVIEW and LabWindows/CVI. The NI TestStand Help includes detailed documentation for the TestStand API.

TestStand Sequence Editor

The sequence editor is a development environment in which you create, edit, execute, and debug sequences and the tests sequences call. Use the sequence editor to access all features, such as step types and process models. Refer to the Process Models section of this tutorial for more information about process models.

You can debug a sequence using the following techniques, similar to how you debug in application development environments (ADEs) such as LabVIEW, LabWindows/CVI (ANSI), and Microsoft Visual Studio:

- Setting breakpoints
- Stepping into, out of, or over steps
- Tracing through program executions
- Displaying variables
- Monitoring variables, expressions, and output messages during executions
- Performing static analysis of sequence files to locate errors and enforce coding guidelines

In the sequence editor, you can start multiple concurrent executions, execute multiple instances of the same sequence, or execute different sequences at the same time. Each execution instance opens an Execution window. In Trace Mode, the Execution window

shows the steps in the currently executing sequence. If the execution suspends, the Execution window shows the next step to execute and provides debugging options.

In the sequence editor, you can fully customize the pane and tab layout to optimize development and debugging tasks. You can also customize the menus, toolbars, and keyboard shortcuts.

User Interfaces

A TestStand User Interface is an application you deploy to a development system or a production station to provide a custom GUI for executing, debugging, or editing sequences. Simple user interfaces might only support running sequences, and custom sequence editors might support editing, running, and debugging sequences.

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

TestStand ships with full-featured Custom Sequence Editor user interfaces. These interfaces can be found by default at `C:\Users\Public\Documents\National Instruments\TestStand 20XX\UserInterfaces\Full-Featured\`, or by navigating to **Start » All Programs » National Instruments » TestStand <Version> » User Interfaces** from the Windows Start Menu.

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

Module Adapters

The TestStand Engine uses module adapters to invoke code modules that sequences call. A code module is a program module from an ADE or programming language and can contain one or more functions that perform a specific test or other action. Module adapters load and call code modules, pass parameters to code modules, and return values and status from code modules. The module adapters support the following

types of code modules:

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

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

TestStand includes the following module adapters:

- LabVIEW Adapter—Calls LabVIEW VIs with a variety of connector panes. The VIs can exist in LLBs or in LabVIEW packed project libraries. You can also call VIs in the context of a LabVIEW project or call LabVIEW class member VIs.
- LabWindows/CVI Adapter—Calls C functions in a DLL with a variety of parameter types.
- C/C++ DLL Adapter—Calls C/C++ functions and static C++ class methods in a DLL with a variety of parameter types. You can call global static methods or static class methods in C++ DLLs. You can create the DLL code module with LabWindows/CVI, Visual Studio, or any other environment that can create a C/C++ DLL, including LabVIEW-built shared libraries.
- .NET Adapter—Calls .NET assemblies written in any .NET-compliant language, such as C# or Microsoft Visual Basic .NET.
- ActiveX/COM Adapter—Creates ActiveX/COM objects, calls methods, and accesses properties of those objects. When you create an object, you can assign the object reference to a variable or property for later use in other ActiveX/COM Adapter steps.
- HTBasic Adapter—Calls HTBasic subroutines without passing parameters directly to a subroutine. TestStand provides a library of CSUB routines that use the TestStand API to access TestStand variables and properties from an HTBasic subroutine.



Note (64-bit TestStand) The HTBasic Adapter is not supported.

- Sequence Adapter—Calls a subsequence in the current sequence file, in another sequence file, or a sequence file on a remote system. You can also make recursive sequence calls.
- Python Adapter—Calls Python modules.

Refer to the following tutorials for more information about using LabVIEW and LabWindows/CVI with TestStand. Refer to the NI TestStand Help for more information about using the LabVIEW and LabWindows/CVI Adapters.

- Calling LabVIEW VIs
- Calling LabWindows/CVI Code Modules

Process Models

Testing a UUT requires more than just executing a set of tests. Usually, the test system must perform a series of operations before and after it executes the sequence that performs the tests. Common operations that define the testing process include identifying the UUT, notifying the operator of pass/fail status, logging results, and generating a report. The set of operations and the flow of execution is called a process model. A TestStand process model is a sequence file you can use to define standard testing operations so you do not have to re-implement the same operations in every sequence file you write.

TestStand includes predefined Sequential, Parallel, and Batch models you can modify or replace. Use the Sequential model to run a test sequence on one UUT at a time. Use the Parallel and Batch models to run the same test sequence on multiple UUTs simultaneously.

You can modify an existing TestStand process model or you can create a custom process model. The ability to modify a process model is essential because the testing process can vary depending on production lines, production sites, or company systems and practices. You can edit a process model in the same way you edit other sequence files. You can also use client sequence files to customize various model operations by overriding the callback sequences process models define.

Entry Points

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

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

Configuration entry points provide an interface for configuring the process model, typically through a GUI. For example, the default Batch model provides the Configure Model Options entry point. This entry point creates the Configure menu item.

TestStand Deployment Utility

Use the TestStand Deployment Utility to create a deployable image or a patch deployment of a TestStand system and an optional installer. The deployable image can contain sequence files, code modules, process models and supporting files, user interface applications, configuration files, and step types and supporting files the TestStand system uses. The installer can contain all files from a deployable image or contain only a subset of files to create a patch for a previously deployed image.

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

Related concepts:

- [Deploying TestStand Systems](#)

- [Calling LabVIEW VIs](#)
- [Calling LabWindows/CVI Code Modules](#)

Related information:

- [TestStand Advanced Architecture Series](#)

Loading and Running Sequences

The TestStand Sequence Editor includes menus, toolbars, windows, and panes you use to load workspace files and edit and run sequence files. Click **Help»Help Topic** to launch the NI TestStand Help for specific information about the active window, tab, or pane in the sequence editor.



Note Completed solution files are located in the <TestStand Public>\Tutorial\Solution directory.

Starting TestStand

You can launch the sequence editor by selecting **Start»NI TestStand**.



Note TestStand prompts you to log in if you open a version of TestStand older than TestStand 2024 Q4. Provide the default user name, `administrator`, in the User Name ring control and leave the Password field empty to log in.

Menu Bar

By default, the menu bar contains the File, Edit, View, Execute, Debug, Configure, Source Control, Tools, Window, and Help menus. The menus are fully customizable. You can create menus with any sequence editor commands you want. Browse the menus in the sequence editor to familiarize yourself with the contents of each menu.

Toolbars

By default, the sequence editor contains a toolbar with shortcuts to commonly used selections from the menu bar. The toolbar is fully customizable. You can create toolbars with buttons for any environment commands you want.

Windows and Panes

The sequence editor contains tabbed windows and panes you can float, dock, resize, and hide. You can customize the pane and tab layout to optimize development and debugging tasks. You can reset the user interface configuration state of the panes in the sequence editor by selecting **View»Reset UI Configuration**, which resets to a default configuration all modifications previously made to menus, toolbars, and the docking state of panes.



Note When you launch the sequence editor for the first time or when you reset the sequence editor configuration, the layout of the panes defaults to the Small Screen Example configuration or the Large Screen Example configuration, depending on the screen resolution. TestStand considers screens with a resolution of 1280 x 1024 or greater to be large. Select **Configure»Sequence Editor Options** to launch the Sequence Editor Options dialog box and select a sequence editor configuration. On the UI Configuration tab, select an example configuration in the Saved Configurations control and click the **Load Selected** button. Click **OK** to close the Sequence Editor Options dialog box.

The Insertion Palette pane in the TestStand Sequence Editor displays items you can insert into a sequence file. The pane contains a Step Types list and a Templates list. The Step Types list displays the step type menu, and the Templates list displays user-defined template sequences, steps, and variables.

Status Bar

The status bar displays the following information for the sequence editor:

- User name of the current user.
- The environment in which the application is running.
- Name of the process model file the current sequence uses. You can double-click in this area to open that process model file.
- Number of steps you have selected.
- Total number of steps in the active sequence.
- File path of the active report after execution.

Organizing a TestStand System with Workspace and Project Files

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

Use project files to organize related files and directories of files in the test system. You can insert any number of files into a project. For example, you can use a project file to organize a sequence file, code module source files, and other supporting files.

A workspace file opens on the Workspace pane, which displays the content of a workspace file. To open any file from the Workspace pane, double-click the file. To organize the files, use the options in the Edit menu to cut, copy, paste, and delete files in projects or project files in the workspace. You can also drag and drop files on the Workspace pane.

You can open only one workspace file at a time. If you have a workspace file open and you try to open or create another, TestStand prompts you to save the workspace file before opening or creating another file.

You must use workspace files to use the source code control (SCC) system features in TestStand. You can also use workspace files to deploy test systems with the TestStand Deployment Utility.

Loading a Workspace File

Complete the following steps to load and view a workspace file.

1. Select **File»Open File** and navigate to the `<TestStand Public>\Tutorial` directory.
2. Select `Tutorial.tsw` and click **Open**.
3. Expand the **Using TestStand** project, which contains all the sequence files used in the tutorials in this tutorial, including the `Solution` directory.

You can leave the workspace file open on the Workspace pane so you can directly open the sequence files for the tutorials in this tutorial, or you can close the Workspace pane and use the options in the File menu to open the sequence files. If you want to close

the Workspace pane, select **File»Close Workspace File** and click **No** if TestStand prompts you to save the changes.

Creating Test Sequences and Sequence Files

A sequence consists of a series of steps. A step can perform many actions, such as initializing an instrument, performing a complex test, or controlling the flow of execution in a sequence.

A sequence file can contain one or more sequences. Sequence files can also contain global variables, which all sequences in the sequence file can access. The sequence editor color-codes sequences by type, such as Execution entry points, Configuration entry points, process model callback sequences, subsequences, and engine callbacks.

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

A sequence can have any number of parameters and local variables. Use parameters to pass data to a sequence when you call the sequence as a subsequence. Use local variables for storing data relevant to the execution of the sequence, storing any other data needed only in the current sequence, maintaining counts, or holding intermediate values.

In the sequence editor, the Variables pane displays all the variables and properties the selected sequence has access to at run time. When you execute a sequence, the Variables pane displays the sequence context for the sequence currently selected on the Call Stack pane. The sequence context contains all the variables and properties accessible in the current execution. Use the Variables pane to examine and modify the values of these variables and properties when an execution is suspended.

The sequence editor opens each sequence file in a separate Sequence File window, which includes the Sequences pane, the Steps pane, and the Variables pane. The Sequences pane lists the name, comment, and requirements values for all sequences

in the sequence file. Use the Sequences pane to select the active sequence to display on the Steps pane, to insert new sequences, and to delete existing sequences from a sequence file.

Loading a Sequence File

Complete the following steps to load and view a sequence file.

1. Select **File»Open File** and navigate to the `<TestStand Public>\Tutorial` directory.
2. Select `Computer.seq` and click **Open**. The `Computer.seq` sequence file is a simulated test of a computer in which you can designate various hardware components to fail. The sequence runs tests implemented as functions in a DLL. TestStand includes examples of DLLs written in LabVIEW, .NET, and LabWindows/CVI.



Note If you left the `Tutorial.tsw` workspace file open from the Loading a Workspace File section of this tutorial, you can open `Computer.seq` from the Workspace pane by double-clicking `Computer.seq` in the list of files.

3. Select **MainSequence** on the Sequences pane.
4. Browse the contents of each pane in the Sequence File window. Ensure that the Sequences pane and the Steps pane are visible when you finish.

Executing a Sequence

You can execute a sequence directly or execute a sequence using a process model Execution entry point. The process model entry point sequence contains a series of steps that specify the high-level flow of an execution.

The TestStand Sequence Editor launches a separate Execution window for each execution you start using the Execute menu. Use the Execution window to view, or trace, steps as they execute, to monitor the values of variables and properties, and to examine test reports when the execution completes.

In Trace Mode, the Execution window shows the steps in the currently executing sequence and indicates the currently executing step. If the execution suspends, the

Execution window shows the next step to execute and provides debugging options.

Before the execution begins, by default, the TestStand Sequence Analyzer analyzes the active sequence file and detects the most common situations that can cause run-time failures. The sequence analyzer prompts you to resolve the reported errors before you execute the sequence file. Carefully review any errors the sequence analyzer returns to prevent run-time failures. If you want to disable the sequence analyzer, click **Debug»Sequence Analyzer»Toggle Analyze File Before Executing**.

Executing a Sequence Directly

Complete the following steps to run `MainSequence` in the `Computer.seq` sequence file directly.

1. Select **Configure»Station Options** to launch the Station Options dialog box. The Execution tab contains options for configuring how an execution behaves. By default, the Enable Tracing option is enabled to specify that the sequence editor displays the progress of an execution with a yellow arrow icon that appears to the left of the currently executing step. This icon is called the execution pointer. The Execution tab also contains a Speed slider control that specifies the tracing speed of the execution. Use this option to slow down the tracing speed so you can observe each step as it executes. By default, the tracing speed is set to fast.
2. Adjust the **Speed** slider control to slow. Click **OK** to close the Station Options dialog box.
3. Select **Execute»Run MainSequence**. The sequence editor opens an Execution window to show the sequence as it executes. The first step in the Setup step group of `MainSequence` launches the Test Simulator dialog box. The Test Simulator dialog box prompts you to designate the computer component, if any, you want to fail during the execution.
4. Enable the **RAM** test option.
5. Click **Done**.
6. Observe the Execution window as it traces through the steps TestStand runs. After the execution completes, the Execution window dims, and the Status column of the `RAM` test contains the value `Failed`.



Note If the tracing speed is too slow or fast, adjust the **Speed** slider control on the Execution tab of the Station Options dialog box.

7. Right-click the Execution window tab and select **Close** from the context menu to close the Execution window.

Executing a Sequence Using the Sequential Process Model

In addition to executing sequences directly, you can execute sequences using an Execution entry point, which is a sequence in a process model file that invokes a test sequence file, typically by calling the `MainSequence` callback in the client sequence file. Executing an Execution entry point performs a series of operations before and after calling `MainSequence` of the sequence file. Common operations of the process model include identifying the UUT, notifying the operator of pass/fail status, logging results, and generating reports.

The Sequential process model includes the following Execution entry points:

- **Test UUTs Execution Entry Point**—The Test UUTs Execution entry point initiates a loop that repeatedly identifies and tests UUTs.
- **Single Pass Execution Entry Point**—The Single Pass Execution entry point tests a single UUT without identifying it. Use the Single Pass Execution entry point when you want to debug tests or determine whether the sequence execution proceeds as you intended.

Complete the following steps to run `MainSequence` in the `Computer.seq` sequence file using the Test UUTs Execution entry point of the Sequential model.

1. Launch the Station Options dialog box.
2. Click the **Model** tab, verify that `SequentialModel.seq` is selected from the Station Model ring control to select the Sequential model as the default process model, and click **OK**.
3. Select **Execute»Test UUTs**. Before executing the steps in `MainSequence`, the process model sequence launches a UUT Information dialog box that prompts you for a serial number.
4. Enter any serial number and click **OK**.
5. Use the options in the Test Simulator dialog box to select any test other than the Video or CPU tests to fail. You can also allow all the tests of the UUT to pass.
6. Click **Done**. Observe the Execution window as the sequence executes. After completing the steps in `MainSequence`, the process model displays a banner that indicates the result of the UUT.

7. Click **OK** to close the UUT Result banner. TestStand generates a report, but does not display the report until you finish testing all the UUTs. TestStand launches the UUT Information dialog box again.
8. Repeat steps 4 through 7 using several different serial numbers.
9. Click **Stop** in the UUT Information dialog box to stop the loop and complete the execution. After the execution completes, TestStand displays the report on the Report pane of the Execution window for all the tested UUTs.
10. Review the test report, which includes the results for each UUT, and select **Window»Close All Windows** to close all the windows in the sequence editor.

Executing a Sequence Using the Batch Process Model

Use the Batch model to control a set of test sockets that test multiple UUTs as a group. For example, if you have a set of circuit boards attached to a common carrier, use the Batch model to ensure that you start and finish testing all boards at the same time. With the synchronization features of the Batch model, you can direct a step that applies to the batch as a whole to run only once per batch instead of once for each UUT. You can also specify whether certain steps or groups of steps cannot run on more than one UUT at a time or whether certain steps must run on all UUTs at the same time. The Batch model generates batch reports that summarize the test results for the UUTs in the batch.

Complete the following steps to run the `BatchUUT.seq` sequence file using the Test UUTs Execution entry point of the Batch model.

1. Open `<TestStand Public>\Tutorial\BatchUUT.seq` and examine the steps and comments in the sequence file to familiarize yourself with the functionality of the sequence.



Note You do not need to change the default process model in the sequence editor for this tutorial. The `BatchUUT.seq` sequence file always uses the Batch model, regardless of the default process model of the sequence editor. Use the Advanced tab of the Sequence File Properties dialog box to specify the process model a sequence file uses.

2. Select **Configure»Model Options** to launch the Model Options dialog box.
3. In the Multiple UUT Settings section, change **Number of Test Sockets** to 2 and enable the **Tile Execution Windows** option. The Number of Test Sockets control

specifies the number of UUTs to test in the batch.

4. In the Batch Settings section, select **Don't Synchronize** from the Default Batch Synchronization ring control and click **OK**.
5. Select **Execute»Test UUTs**. Before executing the steps in `MainSequence`, the process model sequence launches the UUT Information dialog box for the Batch model, which prompts you for a batch serial number and UUT serial numbers for each test socket. You can also disable test sockets in the UUT Information dialog box.
6. Enter any batch serial number and UUT serial numbers. Click **Go**. Review the information in the Batch Model Example dialog box, then click **OK**. TestStand launches several different front panels to indicate the progress of the executions. Click **OK** to continue. After completing the steps in `MainSequence`, the process model launches the Batch Results dialog box, which displays a banner that indicates the result of the UUTs. You can view the batch report and individual UUT reports or start the next batch.
7. Click the **View Batch Report** button. From the pop-up list, you can select Entire File to show all tested batches or Current Only to show the most recently tested batch.
8. Select **Current Only** from the pop-up list. TestStand launches the default application associated with the file extension of the report, such as Internet Explorer for Automatic Test Markup Language (ATML), XML, and HTML reports and Microsoft Notepad for text reports.



Note Enable Internet Explorer to run scripts if the browser launches a prompt. To use another application as an external report viewer, select **Configure»External Viewers** and follow the instructions in the Configure External Viewers dialog box. Refer to the Using External Report Viewers section of Customizing Reports, for more information about using external viewers.

9. Close Internet Explorer and click **Next Batch** in the Batch Results dialog box.
10. Repeat steps 6 through 9 for several different batches.
11. Click **Stop** in the UUT Information dialog box to complete the execution. After the execution completes, TestStand displays test reports on the Report pane of the Execution window for all batches and UUTs.
12. Examine the reports that include results for each batch and UUT.
13. Close all the windows in the sequence editor.

Editing Steps in a Sequence

You can add a step to a sequence, configure the step to call a code module or subsequence, and configure the properties of a step.

The Insertion Palette contains a set of predefined step types you can add to sequences. Step types define the standard behavior and a set of step properties for each step of that type. All steps of the same type have the same properties, but the values of the properties can differ.

When you build sequence files, you can also use the Templates list in the Insertion Palette. Use the Templates list to hold copies of steps, variables, and sequences you reuse during the development of sequence files. For example, you can add a step that calls a specific LabVIEW VI you typically use or a sequence that contains common setup steps, cleanup steps, and local variables.



Note Completed solution files are located in the `<TestStand Public>\Tutorial\Solution` directory.

Adding a New Step

Complete the following steps to add a Pass/Fail Test step to a sequence and configure the step to call a function in a LabWindows/CVI DLL code module by specifying the module adapter to use. TestStand does not need the source code to invoke a DLL code module. Instead, TestStand uses module adapters to determine the type of code module, how to call the code module, and how to pass parameters to the code module.

1. Open `<TestStand Public>\Tutorial\Computer.seq`.
2. Select **File»Save <filename> As** and save the sequence file as `Computer2.seq` in the `<TestStand Public>\Tutorial` directory.



Note When you must save a file, this tutorial specifies the suggested name. If other users will use the tutorial files on the same computer, save the files with unique filenames.

3. Select the **LabWindows/CVI** adapter from the Adapter ring control above the Step Types list to specify the module adapter the step uses. The adapter you select applies only to the step types that can use the module adapter.



Note If you left the `Tutorial.tsw` workspace open after completing the Loading and Running Sequences tutorial, you might need to click the **Insertion Palette** tab to show the Insertion Palette.

When you insert a step in a sequence, TestStand configures the step to use the adapter you selected from the Insertion Palette. The icon for the step reflects the adapter you selected. When you select <None> as the adapter and then insert a step, the new step does not call a code module. Use the General panel on the Properties tab of the Step Settings pane to change the associated adapter after you insert the step.

4. On the Insertion Palette, select **Tests»Pass/Fail Test** and drag the step below the RAM step on the Steps pane to add a Pass/Fail Test step. By default, the name of the new step is Pass/Fail Test.



Note You can also create steps by right-clicking the Steps pane, selecting **Insert Step** from the context menu, and selecting the type of step you want to insert.

Use a Pass/Fail Test step to call a code module that returns a pass/fail determination. After the code module executes, the Pass/Fail Test step evaluates a Boolean expression to determine whether the step passed or failed.

5. Rename the new step `Video Test` by selecting the step on the Steps pane and pressing the <F2> key.
6. Save the changes. Leave the sequence file open for the next tutorial.

Specifying the Code Module

Complete the following steps to specify the code module the step executes and to specify the parameters for a function in the code module for the `Computer2.seq` sequence file you created in the previous tutorial.

1. Select the **Video Test** step and click the **Module** tab of the Step Settings pane.
2. Click the **Browse** button located to the right of the Module control, select `<TestStand Public>\Tutorial\computer.dll`, and click **Open**. When you select a DLL, TestStand reads the type library or exported information of the DLL and lists the functions TestStand can call in the Function ring control.
3. Select **VideoTest** from the Function ring control. TestStand uses the prototype information stored in the type library or the exported information of the DLL to populate the Parameters Table.
4. In the Value Expression column of the result parameter, enter `Step.Result.PassFail`. When TestStand returns from calling the `VideoTest` function during execution, TestStand assigns the value from the **result** parameter to the `Step.Result.PassFail` property of the step.
5. Save the changes. Leave the sequence file open for the next tutorial.

For more information about calling code modules, refer to the following tutorials:

- Calling LabVIEW VIs
- Calling LabWindows/CVI Code Modules

Configuring Step Properties



Note Before you begin this tutorial, select **Configure»Station Options** to launch the Station Options dialog box, click the **Execution** tab, and confirm that the Enable Tracing and Allow Tracing into Setup/Cleanup options are enabled. Click **OK** to close the Station Options dialog box.

Each step in a sequence contains properties. All steps have a common set of properties that determine the following attributes:

- When to load the code module for the step
- When to execute the step
- What information TestStand examines to determine the status of the step
- Whether TestStand executes the step in a loop
- What conditional actions occur after a step executes

Steps can also contain additional properties the step type defines. Use the Properties tab of the Step Settings pane to examine and modify the values of properties of a step.

Complete the following steps to examine and modify step properties in the `Computer2.seq` sequence file you created in the previous tutorial.

1. Select the **Video Test** step on the Steps pane and click the **Properties** tab of the Step Settings pane.
2. Click **Preconditions** on the Properties tab to show the Preconditions panel.

A precondition is a set of conditions for a step that must evaluate to `True` for TestStand to execute the step during the normal flow of execution in a sequence.

3. Complete the following steps to define a precondition so the Video Test step executes only if the Power On step passes.
 - a. Click the **Precondition Builder** button, located to the right of the Precondition Expression control on the Preconditions panel, to launch the Precondition Builder dialog box.
 - b. In the Insert Step Status section, select the **Power On** step from the list of step names for the Main step group and click the **Insert Step Pass** button. The Conditions control now contains the string `PASS Power On`, which indicates that the step executes only if the Power On step passes.
 - c. Click **OK** to close the Precondition Builder dialog box and confirm that the Preconditions panel displays the expression you created.
4. Click **Post Actions** on the Properties tab of the Step Settings pane to show the Post Actions panel, on which you can specify what type of action occurs after the step executes. You can make the action conditional on the pass/fail status of the step or on any custom condition expression.
5. Select **Terminate execution** from the On Fail ring control.
6. Click **Looping** on the Properties tab of the Step Settings pane to show the Looping panel, on which you can configure an individual step to run repeatedly in a loop when the step executes. Use the Loop Type ring control to select the type of looping for the step.

TestStand determines the final status of the step based on the number of passes, failures, or loop iterations that occur.

7. On the Looping panel, enter the following values into the corresponding controls. When you change the value of a property, TestStand shows the new value in bold to indicate that it differs from the default value.

- **Loop Type**—Fixed number of loops
- **Number of Loops**—10
- **Loop result is Fail if**—<80%

Using these settings, TestStand executes the Video Test step 10 times and sets the overall status for the step to `Failed` if fewer than 8 of the 10 iterations pass.

8. Confirm that the Settings column on the Steps pane of the Sequence File window shows that the Video Test step contains Loop, Precondition, and Post Action settings. Use the tooltip in the Settings column to verify the values for each setting.
9. Save the changes and select **Execute»Single Pass**.
10. Select the **Video** test to fail and click **Done**. The sequence immediately terminates after calling the Video Test step 10 times in a loop. TestStand records the result of each loop iteration in the report.
11. Close the Execution window.

Calling Subsequences

Use the Sequence Call step to call another sequence in the current sequence file or in another sequence file.

Complete the following steps to add a Sequence Call step to the `Computer2.seq` sequence file you created in the previous tutorial.

1. Insert a Sequence Call step after the Power On step on the Steps pane and rename the step `CPU Test`.
2. On the Module tab of the Step Settings pane for the CPU Test step, click the **Browse** button located to the right of the File Pathname control and select `<TestStand Public>\Tutorial\CPU.seq` to specify the sequence the step invokes.
3. Select **MainSequence** from the Sequence ring control on the Module tab of the Step Settings pane.
4. Save the changes and double-click the **CPU Test** step to open the sequence and show the `MainSequence` of `CPU.seq` in a new Sequence File window.
5. Select **Execute»Run MainSequence**. Examine the execution of `CPU.seq` so you can recognize it later when you execute the `Computer2.seq` sequence file that calls `CPU.seq`.
6. Close the Execution window and the `CPU.seq` Sequence File window.
7. In the `Computer2.seq` Sequence File window, select **Execute»Single Pass**.

8. Select a test to fail and click **Done**. After the sequence executes, review the test report. TestStand logs the results of the steps in the subsequence in addition to the steps from the parent sequence.
9. Close the Execution window. Leave the `Computer2.seq` sequence file open for the next tutorial.

Using Step Templates

The Insertion Palette contains the Step Types list and the Templates list. Use the Templates list to hold copies of steps, variables, and sequences you reuse during the development of sequence files. For example, you can add a step that calls a specific LabVIEW VI you typically use or a sequence that contains common setup steps, cleanup steps, and local variables.

Drag steps from the Steps pane, variables from the Variables pane, and sequences from the Sequences pane and drop them on the Templates list to add steps, variables, or sequences to the Templates list. Use the context menu to rename, copy, paste, delete, import, and export the items in the Templates list. You can drag and drop items to rearrange the list. Select **Insert Folder** from the context menu to add folders to the list. TestStand stores the settings for the Templates list, including any modifications to the list, in the `<TestStand Application Data>\Cfg\Templates.ini` file.

You cannot directly modify templates. Drag a sequence, step, or variable from the Templates list to a sequence file to edit the item. Then, drag the item back to the Templates list and delete the original item from the Templates list.

Inserting a Template Step

Complete the following steps to import a set of template steps to the Insertion Palette, insert a step from the Templates list in a sequence, and add a new step to the Templates list in the `Computer2.seq` sequence file you created in the previous tutorial.

1. Right-click the **Steps** folder in the Templates list of the Insertion Palette and select **Import** from the context menu.
2. Navigate to the `<TestStand Public>\Tutorial` directory, select `Tutorial Templates.ini`, and click **Open**. The sequence editor adds a Tutorial folder under the Steps folder.

3. Expand the **Tutorial** subfolder of the Steps folder to view the following imported template steps:
 - **Retry if Previous Step Fails**—A Message Popup step that prompts you to retry the previous step on failure.
 - **Open Notepad**—A Call Executable step configured to launch the Microsoft Notepad application.
 - **Output Message**—A statement step that logs an output message in which the logged text is the step name.
 - **Waveform Popup**—A step that calls the WaveformGraphPopup function in the LabWindows/CVI user interface library.
4. In the Templates list, select **Retry if Previous Step Fails**, drag the step below the Power On step on the Steps pane, and rename the step `Retry Power On`.
5. Save the changes and select **Execute»Single Pass**.
6. Select the **Power On** test to fail and click **Done**. TestStand executes the Retry Power On step and launches the Step 'Power On' Failed dialog box because the precondition for the step determined that the previous Power On step failed.
7. Click **Retry** in the Step 'Power On' Failed dialog box to execute the Power On step again. The Power On step fails again, and TestStand launches the Step 'Power On' Failed dialog box again.
8. Click **Continue** in the Step 'Power On' Failed dialog box to continue the execution.
9. Review the report. Notice that the Power On step executed twice, and the second call to Retry Power On continued the execution.
10. Close the Execution window. Leave the sequence file open for the next tutorial.

Creating a Template Step

Complete the following steps to update the Retry Power On step so that it automatically selects the **Continue** button if you do not respond to the prompt in the `Computer2.seq` sequence file you created in the previous tutorial.

1. In the Sequence File window, select the **Retry Power On** step on the Steps pane.
2. Click the **Text and Buttons** tab of the Step Settings pane.
3. In the Button Options section, select **Button 2** from the Timeout Button ring control.
4. Enter `20` in the Time To Wait control to instruct the step to wait for 20 seconds before continuing. This technique is useful if an operator is not present to acknowledge a non-critical message during testing.
5. Save the changes and select **Execute»Single Pass**.

6. Select the **Power On** test to fail and click **Done**. TestStand executes the Retry Power On step and launches the Step 'Power On' Failed dialog box. Do not take any action in this dialog box. When the timeout reaches zero, the execution continues as if you clicked **Continue**.
7. In the Sequence File window, drag the Retry Power On step into the Tutorial folder in the Templates list.
8. Rename the new template step `Timeout Retry`. You can use the new template step in subsequent development. The sequence editor automatically saves the templates list when you shut down the sequence editor.
9. Close the `Computer2.seq` sequence file and the Execution window.
10. You no longer need the Tutorial folder in the Templates list, and you can delete the folder if you want. Right-click the **Tutorial** folder in the Templates list and select **Delete** from the context menu to remove the folder.

Related concepts:

- [Calling LabVIEW VIs](#)
- [Calling LabWindows/CVI Code Modules](#)

Debugging Sequences

You can debug sequences by tracing, setting breakpoints and conditional breakpoints, stepping through code, and including watch expressions. Refer to Using the Watch View Pane for more information about watch expressions.



Note Completed solution files are located in the `<TestStand Public>\Tutorial\Solution` directory.

Step Mode Execution



Note Before you begin this tutorial, select **Configure»Station Options** to launch the Station Options dialog box, click the **Execution** tab, and confirm that the Enable Tracing and Allow Tracing into Setup/Cleanup options are enabled. Click **OK** to close the Station Options dialog box.

Setting a Breakpoint

Complete the following steps to set a breakpoint in a sequence file.

1. Open `<TestStand Public>\Tutorial\Computer2.seq`, which you created in the previous tutorial.
2. Select **File»Save <filename> As** and save the sequence file as `Computer3.seq` in the `<TestStand Public>\Tutorial` directory.
3. Select **Execute»Break on First Step** to suspend an execution on the first step TestStand executes. A checkmark appears to the left of the menu option to indicate that you enabled this option.
4. Expand the **Cleanup** step group on the Steps pane. Steps in the Cleanup step group execute regardless of whether the sequence completes successfully or whether a run-time error occurs in the sequence. If a step in the Setup or Main step group causes a run-time error to occur or if the operator terminates the execution, the flow of execution stops and jumps to the Cleanup step group. Steps in the Cleanup group always run even when some of the steps in the Setup group do not run. When a step in the Cleanup group causes a run-time error, execution continues to the next step in the Cleanup group.
5. Insert a Message Popup step in the Cleanup step group and rename the step Cleanup Message.
6. On the Text and Buttons tab of the Step Settings pane, enter "I am now in the Cleanup Step Group.", including the quotation marks, in the Message Expression expression control.



Note

- You must enclose literal strings in double quotation marks ("&mlr;") in any TestStand expression field.
- For this example, use the default values for all the settings on the Options tab and the Layout tab of the Step Settings pane.

7. Save the changes. Leave the sequence file open for the next tutorial.

Stepping through a Sequence File

When you suspend an execution, you can step through the sequence using the Step Into, Step Over, and Step Out buttons on the Execution window.

You can find these and other debugging tools in the Debug menu of the sequence editor.

Complete the following steps to step through an execution of the `Computer3.seq` sequence file you created in the previous tutorial.

1. Select **Execute»Run MainSequence**. After the execution starts, the sequence editor immediately suspends at the first step of the sequence because you enabled the Break on First Step option in step 3 of the previous tutorial. The Execution window tab includes an execution pointer icon to indicate the running state of the execution. When the execution suspends, you can step through the sequence using the Step Into, Step Over, and Step Out commands at the top of the Execution window.
2. Click the **Step Over** button to execute the Display Dialog step, which launches the Test Simulator dialog box.
3. Select the **RAM** test to fail and click **Done**. After the Test Simulator dialog box closes, the sequence editor suspends the sequence execution at the end of the Setup step group on `<End Group>`.
4. Insert a breakpoint at the CPU Test step by clicking to the left of the step icon. In the Execution window, a dark red icon appears to the left of the CPU Test step to indicate the breakpoint.



Note To specify a conditional breakpoint, right-click the breakpoint icon and select **Breakpoint»Breakpoint Settings** to launch the Breakpoint Settings dialog box, in which you can specify an expression that must evaluate to `True` to activate the breakpoint. Click **OK** to close the Breakpoint Settings dialog box. Conditional breakpoints use an orange icon in the Sequence File and Execution windows. Disabled breakpoints use a gray icon. When you globally disable all breakpoints, the breakpoints use a white icon. Refer to Using Variables and Properties for more information about expressions.

5. Click the **Resume** button to continue the execution. The sequence editor suspends the execution on the `CPU Test` step.
6. Click the **Step Into** button to step into the `MainSequence` subsequence in `CPU.seq`, and click the **Call Stack** pane. If the Call Stack pane is not visible, click **View»Debug»Call Stack** to display it. Sequence call steps are similar to subVIs in

LabVIEW and function or method calls in C/C++. 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 wait for nested subsequences to complete is called the call stack. The first item in the call stack is the most-nested sequence invocation. The Call Stack pane displays the call stack for the execution thread currently selected on the Threads pane. A yellow pointer icon appears to the right of the most nested sequence invocation while the sequence executes, as shown in Figure 4-2.

7. Select each sequence on the Call Stack pane to view each sequence invocation. Return to the most nested sequence invocation in the call stack, `CPU.seq`.
8. Click the **Step Over** button to step through the `CPU.seq` subsequence one step at a time. Before you reach the end of the `CPU.seq` sequence, click the **Step Out** button. TestStand resumes the execution through the end of the current sequence and suspends the execution at the next step of the calling sequence or breakpoint, whichever comes first.
9. Continue stepping through the `Computer3.seq` sequence by clicking the **Step Over** button until the `Cleanup Message` step you added to the Cleanup step group executes. You must step over each of the 10 loops of the Video Test step.
10. Click **OK** to close the Cleanup Message dialog box and click the **Step Over** button to complete the execution. The Execution window dims when the execution completes. Do not close the Execution window.
11. Click the **Restart** button to rerun the execution. The Execution window must be the active window to restart the execution.
12. After the sequence editor suspends the execution on the first step, click the **Terminate Execution** button. TestStand launches the Cleanup Message dialog box even though you terminated the sequence execution. When an operator or run-time error terminates the execution, TestStand proceeds immediately to the steps in the Cleanup step group. Click **OK** to close the Cleanup Message dialog box. The Execution window dims when the execution completes.
13. Click the **Restart** button to rerun the execution. The Execution window must be the active window to restart the execution.
14. After the sequence editor suspends the execution on the first step, select **Debug»Abort All (no cleanup)**. The execution of the sequence immediately stops, and TestStand does not execute any steps in the Cleanup step group.
15. Save the changes and close all the windows in the sequence editor.

Related concepts:

- [Using Variables and Properties](#)

Using Variables and Properties

You can create and use variables and properties and monitor the values of the variables and properties.

You can define the following types of variables to share data among steps of a sequence or among several sequences:

- Local variables store data relevant to only the current sequence. Only steps within the sequence that defines the local variable can access these variables.
- Sequence file global variables store data relevant to the entire sequence file. Each sequence and step in the sequence file can directly access these global variables.
- Station global variables persist across different executions. The TestStand Engine maintains the value of station global variables in a file on the computer on which TestStand is running.

In the sequence editor, the Variables pane displays all the variables and properties the selected sequence can access at run time. In the Execution window, the Variables pane displays the sequence context for the sequence invocation currently selected on the Call Stack pane. The sequence context contains all the variables and properties the steps in the selected sequence invocation can access. Use the Variables pane to examine and modify the values of these variables and properties when the sequence is in a suspended state, such as when paused at a breakpoint.



Note Completed solution files are located in the `<TestStand Public>\Tutorial\Solution` directory.

Creating Local Variables

Complete the following steps to create and use local variables. You can apply the concepts you learn in this tutorial to sequence file global and station global variables.

1. Open `<TestStand Public>\Tutorial\Computer2.seq`, which you

created in Editing Steps in a Sequence.

2. Select **File»Save <filename> As** and save the sequence file as `Computer4 .seq` in the `<TestStand Public>\Tutorial` directory.
3. Click the **Variables** pane in the Sequence File window. Expand the **Locals** item to view the local variables currently defined for `MainSequence`. Each sequence includes a `ResultList` local variable, which is an empty array of container properties. TestStand uses this variable to store step results for result processing.
4. On the Variables pane, right-click **Locals**, select **Insert Local»Number** from the context menu to insert a new numeric local variable, and rename the local variable `LoopIndex`.
5. Complete the following steps to insert and configure a For step.
 - a. Click the **Steps** pane in the Sequence File window and insert a **Flow Control»For** step below the **Retry Power On** step. Notice that TestStand also adds an **End** step below the For step.
 - b. Drag the **End** step below the **RAM** step. TestStand automatically indents the **Retry Power On**, **CPU Test**, **ROM**, and **RAM** steps between the For and End steps.
 - c. Click the **For** step and click the **For Loop** tab of the Step Settings pane.
 - d. Enter 5 in the **Number of Loops** control.
 - e. For the Loop Variable control, click the **Expression Browser** button to launch the Expression Browser dialog box, in which you can interactively build an expression by selecting from lists of variables, properties, operators, functions, and the TestStand API. TestStand supports all applicable expression operators and syntax you can use in C, C++, Java, and Visual Basic .NET. You can also call the TestStand API directly from within expressions. All TestStand controls that accept expressions provide context-sensitive editing features, such as drop-down lists, syntax checking, and expression coloring to help you create expressions. At any point while editing an expression, you can press `<Ctrl-Space>` to show a drop-down list of valid expression elements.
 - f. Expand the **Locals** item on the Variables/Properties tab of the Expression Browser dialog box. Each item in the top section of the Variables/Properties tab is a property or variable of TestStand.
 - g. Select the **LoopIndex** variable under the **Locals** property and click the **Insert** button. The Expression Browser enters `Locals . LoopIndex` in the Expression control.



Note To refer to a subproperty, use a period to separate the name of the property from the name of the subproperty. For example,

reference the `LoopIndex` subproperty of the `Locals` property as `Locals.LoopIndex`.

- h. Click the **Check Expression for Errors** button to verify that the expression contains valid syntax.
- i. In the Expression Browser dialog box, click **OK** to return to the For Loop tab of the Step Settings pane. The Loop Variable control now contains the `Locals.LoopIndex` expression. Notice that the Custom Loop section shows the expressions TestStand uses when executing the For step when using a fixed number of loops.
6. Select **Execute»Break on First Step** to remove the checkmark that appears to the left of the menu item and disable this option, which you enabled in the Step Mode Execution section of Debugging Sequences.
7. Save the changes and select **Execute»Single Pass**.
8. Click **Done** in the Test Simulator dialog box.
9. After the sequence executes, review the test report, which shows that TestStand executed the steps within the loop (Retry Power On, CPU Test, ROM, and RAM) five times.
10. Close the Execution window. Leave the sequence file open for the next tutorial.

Using the Execution Window Variables Pane

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

For each active sequence, TestStand maintains a sequence context that contains references to the run-time copy of the sequence so you can access all the objects, variables, and properties that relate to the execution of the sequence.

The contents of the sequence context vary depending on the currently executing sequence and step, the location of the active sequence in the call stack, and the identity of the execution in which the active sequence resides. Depending on the current state of execution, sequence context subproperties might not exist. When a property exists, the contents of the property can vary.

The Variables pane displays the sequence context for the sequence invocation

currently selected on the Call Stack pane. The sequence context contains all the variables and properties the steps in the selected sequence invocation can access. Use the Variables pane to examine and modify the values of these variables and properties when the sequence is in a suspended state, such as when paused at a breakpoint.

Table 5. Top-level Properties in the Sequence Context

Sequence Context Subproperty	Description
Locals	Contains local variables in the current sequence. Only the current sequence can access these variables.
Parameters	Contains parameter variables in the current sequence. Only the current sequence or calling sequences can access these variables.
FileGlobals	Contains file global variables in the current sequence file. All the sequences in the current sequence file can access these variables.
StationGlobals	Contains station global variables on the computer. Any sequence on the current computer can access these variables. Station global variables are stored on disk and the values persist even after you close TestStand.
ThisContext	Contains a reference to the current sequence context. You typically use this property to pass the entire sequence context as an argument to a subsequence or a step code module.
RunState	Contains properties that describe the current state of execution.
Step	Contains the properties in the currently executing step. The Step property exists only while a step executes. The property does not exist when the execution is between steps, such as at a breakpoint.

Complete the following steps to use the Variables pane of the Execution window to examine the value of the LoopIndex variable while TestStand executes the `Computer4.seq` sequence file you created in the previous tutorial.

1. Insert a breakpoint at the End step associated with the For loop you created in the previous tutorial.
2. Select **Execute»Single Pass**.
3. Click **Done** in the Test Simulator dialog box. The execution suspends on the End step.
4. Click the tab for the Variables pane of the Execution window and expand the

Locals section.

5. Select the **LoopIndex** property. The numeric value of `LoopIndex` is 0.
6. Click the tab for the Steps pane of the Execution window and click the **Resume** button on the Debug toolbar. The execution resumes and suspends at the End step again.
7. Click the tab for the Variables pane again. The value of `Locals.LoopIndex` is now 1. Leave the execution in the suspended state for the next tutorial.

Docking the Variables Pane

The sequence editor contains tabbed windows and panes you can float, dock, resize, and hide. Some panes, such as the Variables pane, must stay attached to the associated Sequence File or Execution window.

Complete the following steps to display the Steps and Variables panes at the same time.

1. In the Execution window, click the tab for the Variables pane and drag it to the right using the tab, not the title bar. As you drag the pane from the current location, the sequence editor detaches the pane and displays docking guides. The docking guides show you where you can place the pane. As you move the mouse over a docking guide, the sequence editor highlights where the pane will relocate when dropped.
2. Drop the pane on the right-most docking guide, so the pane appears to the right of the Steps panes.
3. Click the tab for the Steps pane and click the **Resume** button on the Debug toolbar. The execution resumes and suspends at the End step again. The value of `Locals.LoopIndex` on the Variables pane is now 2.
4. Drag the title bar of the Variables pane and drop it on the center docking guide, to return the Variables pane to the previous location. Notice that the tab for the Variables pane now appears to the left of the tab for the Steps pane instead of to the right.
5. Click the tab for the Variables pane, drag it to the right, and drop it on the Steps pane tab to adjust the order of the tabs. Leave the execution in the suspended state for the next tutorial.



Note Select **View»Reset UI Configuration** at any time to restore the

panes to the original state.

Using the Watch View Pane

The Watch View pane of the Execution window displays the values of watch expressions you enter. The values in the Watch View pane update when execution suspends at a breakpoint. When you enable tracing, the sequence editor also updates the values after each step executes. The Watch View pane highlights viewed and changed watch expression values in red text.

Enter watch expressions to monitor values of variables and properties as you trace or step through a sequence. You can drag individual variables or properties from the Variables pane to the Watch View pane. When you specify a container property, array property, or a `PropertyObject` reference as a watch expression, you can use the Watch View pane to expand the subproperties to view the values.

Complete the following steps to create a watch expression that uses the `LoopIndex` property.

1. Select the **LoopIndex** property on the Variables pane of the Execution window and drag the property to the Watch View pane. The value of the `LoopIndex` watch expression is 2.
2. Edit the `LoopIndex` watch expression directly in the Watch Expression column on the Watch View pane to change the expression to the following:

```
Str (Locals.LoopIndex * 20) + "%"
```

3. Click the tab for the Steps pane of the Execution window and click the **Resume** button on the Debug toolbar. The execution resumes and suspends at the End step again. The value of the watch expression changes from 40% to 60%.



Note The watch expression pane might display an **Error in argument** message before displaying 60% because the watch expression is not valid during a subsequence execution.

4. Click the breakpoint icon to the left of the End step in the Execution window to remove the breakpoint.

5. Resume and complete the execution.
6. Close the Execution window.
7. Select **Debug»Breakpoints/Watches** to launch the Edit Breakpoints/Watch Expressions dialog box, in which you can review and edit the breakpoints and watch expressions associated with the current workspace. TestStand saves and reloads breakpoints and watch expressions using an options file associated with the current workspace or a default options file when no workspace is loaded.
8. Review the contents of the Breakpoints and Watch Expressions tabs in the Edit Breakpoints/Watch Expressions dialog box, delete the breakpoint and expression you created for this tutorial, and click **Done** to close the dialog box.
9. Close the sequence file.

Related concepts:

- [Editing Steps in a Sequence](#)
- [Debugging Sequences](#)

Using Callback Sequences

Callbacks are sequences TestStand calls under specific circumstances. You can create new callbacks or you can override existing callbacks to customize the operation of the test station.



Note Completed solution files are located in the <TestStand Public>\Tutorial\Solution directory.

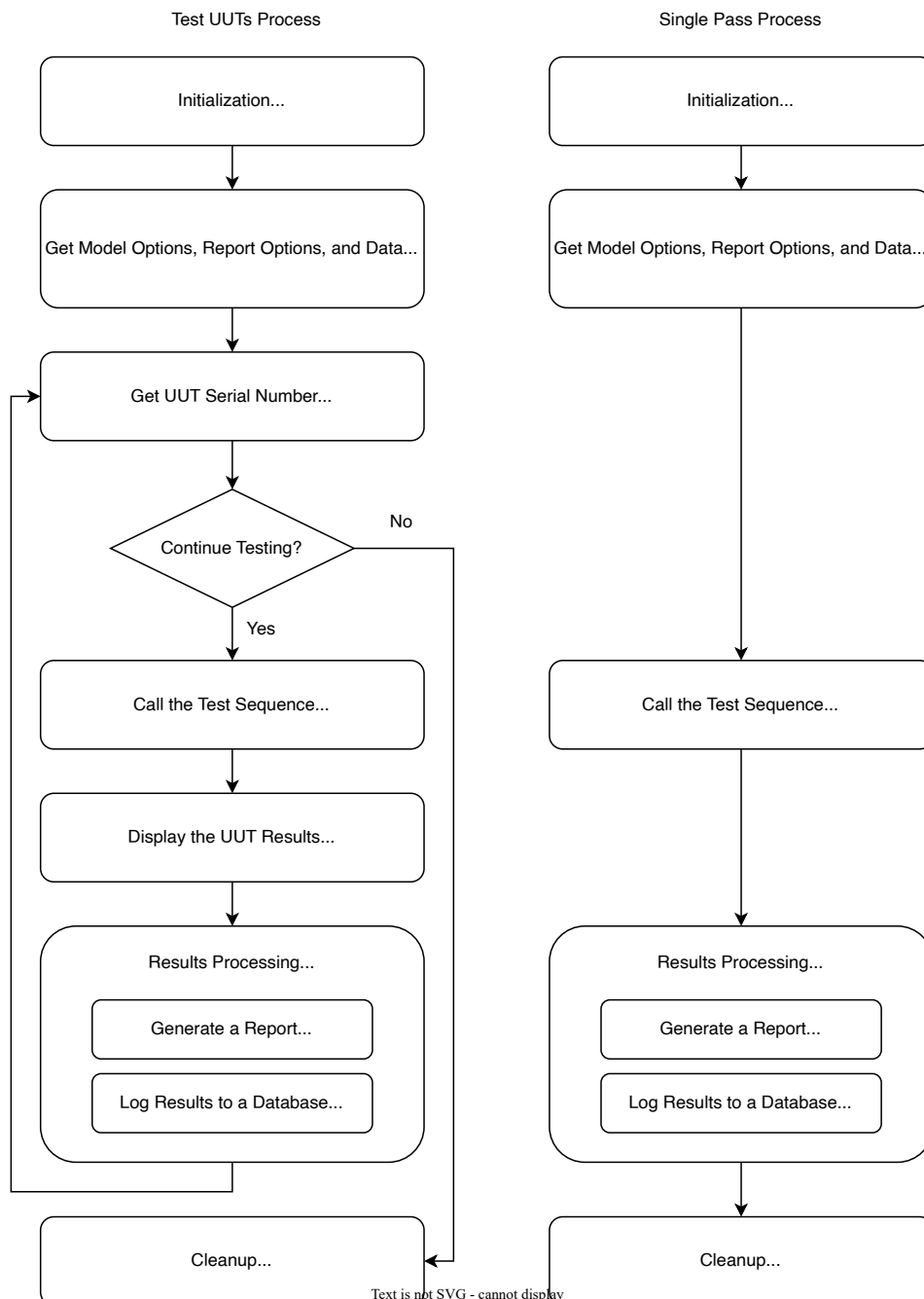
Process Model Callbacks

The process models contain sequences that define the operations TestStand performs before and after testing a UUT. You can use an Execution entry point, such as Test UUTs or Single Pass, to invoke sequences within a process model to execute a sequence in a sequence file you create. The process models also contain Model callbacks, which are sequences you can override to customize the behavior of a process model without editing the process model directly.

For example, the process models define a PreUUT callback sequence that prompts the user for a serial number before the test sequence executes. In most cases, this default

behavior is sufficient. You can override the default PreUUT callback in a sequence file you create. When you override the default PreUUT callback and you use the Test UUTs or Single Pass Execution entry point to execute a sequence file, the process model calls the custom PreUUT callback in the sequence file instead of the default PreUUT callback in the process model.

The following figure shows the actions the default Sequential process model performs using callback sequences within the Test UUTs and Single Pass Execution entry points.



You can modify the process models or replace the models entirely to alter the behavior of the process models for all sequences.



Note To modify the installed process models or to create a new process model, copy all the process model files from the `<TestStand>\Components\Models\TestStandModels` directory to the `<TestStand Public>\Components\Models\TestStandModels` directory and make changes to the copy. Use the following guidelines when you copy installed files to modify:

- You must rename the files after you modify them if you want to create a separate custom component. You must register any new or renamed ActiveX components.
- You do not have to rename the files after you modify them if you only want to modify the behavior of an existing component.
- If you do not rename the files and you use the files in a future version of TestStand, changes NI makes to the component might not be compatible with the modified version of the component.
- Storing new and customized files in the `<TestStand Public>` directory ensures that installations of the same version of TestStand do not overwrite the customizations and ensures that uninstalling TestStand does not remove the files you customize.

Viewing Process Model Callbacks

Complete the following steps to examine the Model callbacks in the Sequential process model, which is the default process model TestStand uses to execute sequences.

1. Open `<TestStand>\Components\Models\TestStandModels\SequentialModel.seq`.
2. On the Sequences pane, select **Test UUTs**, which is the Execution entry point TestStand executes when you select **Execute»Test UUTs**. The Main step group of the Test UUTs sequence calls several callback sequences, including the following:
 - PreUUTLoop callback
 - PreUUT callback

- MainSequence callback
 - PostUUT callback
 - PostUUTLoop callback
3. Double-click the **PreUUT Callback** step, not the PreUUTLoop Callback step, to view the PreUUT callback sequence, which calls the DoPreUUT sequence.
 4. On the Sequences pane, select the **DoPreUUT** sequence. If the DoPreUUT Properties dialog box launches, click **OK** to close it. The DoPreUUT sequence includes an IdentifyUUT step and a Set Serial Number step.



Note If you double-click the Call DoPreUUT step on the Steps pane of the PreUUT Callback sequence to try to open the DoPreUUT sequence, TestStand returns an error that it cannot open the sequence because TestStand evaluates an expression at run time to determine the sequence file path.

5. Right-click the **IdentifyUUT** step and select **Run Selected Steps** from the context menu to launch the UUT Information dialog box, which is similar to the dialog box that launches when you execute a sequence using the Test UUTs Execution entry point. You can override the PreUUT callback with a custom callback to change the way TestStand obtains a UUT serial number, such as reading the serial number from a bar code instead.
6. Click **OK** in the UUT Information dialog box.
7. Close the Execution window.
8. On the Sequences pane, select the **Test UUTs** sequence.
9. Double-click the **PreUUTLoop Callback** step to open the PreUUTLoop callback sequence, which is empty because it is a placeholder. If you want to add steps that execute before the UUT loop, you can complete the steps in the section to override this callback.
10. Close `SequentialModel.seq` and decline any prompts to save the changes.

Overriding a Process Model Callback

You can customize the functionality of a process model without making changes to the model itself. When you override a callback, the process model invokes the callback you create in a sequence file instead of the default callback in the process model.

Complete the following steps to override the default PreUUTLoop callback sequence in the Sequential model by creating a PreUUTLoop callback sequence in a client

sequence file. Use this technique when you want to perform a task only once before operating on multiple UUTs, such as initializing hardware.

1. Open `<TestStand Public>\Tutorial\Computer4.seq`, which you created in Using Variables and Properties.
2. Select **File»Save <filename> As** and save the sequence file as `Computer5.seq` in the `<TestStand Public>\Tutorial` directory.
3. Select **Edit»Sequence File Callbacks** to launch the Sequence File Callbacks dialog box.
4. Enable the checkbox for the **PreUUTLoop** callback. Click **OK** to close the Sequence File Callbacks dialog box. The sequence editor creates a new empty callback sequence in the sequence file. Now, when you start an execution using an Execution entry point, TestStand calls the callback in the sequence file instead of the callback sequence in the Sequential process model file. The PreUUTLoop callback sequence becomes the selected sequence on the Sequences pane and opens in the Sequence File window.
5. Insert a Message Popup step in the Main step group of the PreUUTLoop callback sequence and rename the new step `Pre UUT Loop Callback Message`.
6. Enter the literal string `"Now in the Pre UUT Loop Callback"` in the Message Expression control.
7. Save the changes and select **Execute»Test UUTs**. TestStand launches the Pre UUT Loop Callback Message dialog box.
8. Click **OK** to close the Pre UUT Loop Callback Message dialog box. TestStand launches the UUT Information dialog box from the PreUUT callback sequence in the Sequential model. Enter a serial number and click **OK**.
9. Run through several iterations of the sequence. TestStand launched the Pre UUT Loop Callback Message dialog box only once at the very beginning of the execution because the PreUUTLoop callback executes before the loop, and the PreUUT callback executes within the loop.
10. Click **Stop** in the UUT Information dialog box.
11. Close all the windows in the sequence editor.

The Parallel and Batch process models include similar callbacks that you can override.

Related concepts:

- [Using Variables and Properties](#)

Adding Users and Setting Privileges

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

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



Note Completed solution files are located in the <TestStand Public>\Tutorial\Solution directory.

Adding a New User



Note This tutorial assumes you are currently logged in to TestStand as the `administrator` user. If you are not logged in as administrator, select **File»Login** and select **administrator** from the User Name ring control. Leave the Password field empty and click **OK**.

Complete the following steps to add a new user.

1. Click the **User Manager** button on the toolbar to open the User Manager window, which shows all the users and groups configured on the test station.
2. Right-click the **Users** item, select **Insert User** from the context menu, and enter a name to create a new user.
3. Right-click the new user you just added, select **Properties** from the context menu to launch the <User> Properties dialog box, and complete the following steps.
 - a. Confirm that the User Name control displays the new name you entered.
 - b. Enter the full name in the Full Name control.
 - c. Enter a password in the Password and Confirm Password controls.
 - d. Select **Operator** in the Group Privileges control.

- e. Click **OK** to close the User Properties dialog box.
4. Save the changes to the user manager.
5. Select **File»Login** to launch the Login dialog box. The User Name ring control now includes the new user you just added.
6. Select the user you just created, enter the appropriate password, and click **OK**.
7. Open `<TestStand Public>\Tutorial\Computer.seq`.
8. Select the **Execute** menu and notice that the Single Pass and Run MainSequence options of the Execute menu are disabled because the user you just created does not have the privileges to execute them.
9. Right-click the **Steps** pane to insert a new step. The Insert Step menu command is also disabled because the user privileges have changed.
10. Select **File»Login** and select **administrator** from the User Name ring control. Leave the Password field empty and click **OK**.

Creating a New Group

You can use the user manager to modify the default groups and to create new groups that define a combination of appropriate privileges.

The default Operator, Technician, Developer, and Administrator groups define a set of privilege settings for the new user to inherit. By default, the Operator group grants a user the privilege to execute, terminate, and abort sequences but does not grant the privilege to create or debug sequences.

Complete the following steps to create a new group.

1. Open the User Manager window.
2. Expand the **Groups** item to show the four default groups.
3. Right-click the **Operator** group and select **Copy** from the context menu.
4. Right-click the **Groups** item and select **Paste** from the context menu.
5. Rename the new group `Senior Operator`. The new group is identical to the Operator group except for the name.

Modifying Privileges

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

- **Operate**—Privileges for executing sequences and terminating and aborting executions.
- **Debug**—Privileges for controlling execution flow, executing manual and interactive executions, and editing station global variables and run-time variables.
- **Develop**—Privileges for editing and saving sequence files, editing and saving workspace files, and using source code control.
- **Configure**—Privileges for editing process model files and configuring station options, users, adapters, application settings, and report, database logging, and model options.
- **Custom**—Custom privileges you define. Customize the `NI_UserCustomPrivileges` data type to add new privileges.

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

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



Note TestStand stores the privilege categories as subproperties of the `Privileges` property. The `Privileges` property also includes a Boolean `GrantAll` subproperty. The property `Privileges.GrantAll` applies to all privilege categories. When you set the `GrantAll` property to `True`, the user or group has all privileges. You must set the `GrantAll` property to `False` to honor privilege settings within each privilege category.

Complete the following steps to modify the default privileges for the group you created in the previous section of this tutorial.

1. Expand the **Senior Operator** item and expand the **Privileges** property.
2. Expand the **Debug** item, which is a property that contains Boolean subproperties. Use the **Value** column ring control to change the `SinglePass` property under **Debug** to `True`.
3. Complete the following steps to add the user you created in the Adding a New User section of this tutorial to the Senior Operator group.
 - a. Right-click the user you previously created under Users and select **Properties** from the context menu to launch the <User> Properties dialog box.
 - b. Disable **Operator** in the Group Privileges control and enable **Senior Operator** instead.
 - c. Click **OK** to close the User Properties dialog box.
4. Save the changes to the user manager.
5. Select **File»Login** to launch the Login dialog box. Select the user you previously created, enter the appropriate password, and click **OK**.
6. Select the `Computer.seq` window.
7. Select the **Execute** menu. Notice that the Single Pass option is now available, but the Run MainSequence option is disabled because the user you created does not have the privilege to execute sequences without a Model entry point.
8. Close all the windows in the sequence editor.
9. Select **File»Login** and select **administrator** from the User Name ring control. Leave the Password field empty and click **OK**.

Refer to the `Creating & Deleting Users Using API.seq` example located in the `<TestStand Public>\Examples\TestStand API\Creating & Deleting Users Using APIs` directory for information about how to use the TestStand API to add and remove users programmatically.

Interactive Executions

When you run steps in interactive mode, you can execute specific steps in a sequence. Use the Interactive Executions section on the Execution tab of the Station Options dialog box to control if an interactive execution records results, runs steps in the Setup and Cleanup step groups, and evaluates preconditions. The options in the Interactive Executions section also determine how TestStand handles step failures, errors, and branching during interactive executions.



Note Completed solution files are located in the <TestStand Public>\Tutorial\Solution directory.

Running Selected Steps as a Separate Execution

Complete the following steps to run selected steps from a Sequence File window as a separate execution.

1. Open <TestStand Public>\Tutorial\Computer.seq.
2. Insert a breakpoint at the Power On step.
3. Press <Ctrl> and select the **Power On,ROM, and ROM Diagnostics** steps.
4. Select **Execute»Run Selected Steps** to start a root interactive execution. By default, when you run selected steps from a Sequence File window, TestStand also executes the Setup and Cleanup step groups. You enable or disable the Run Setup and Cleanup option on the Execution tab of the Station Options dialog box to control whether TestStand runs the Setup and Cleanup step groups as part of the root interactive execution.
5. Click **Done** in the Test Simulator dialog box. The execution stops at the Power On step breakpoint. The execution pointer for the interactive execution is a narrow yellow arrow.
6. Step over twice to step through the execution until you reach the ROM Diagnostics step. Only the steps you selected execute. TestStand dims the other steps.
7. Resume and complete the execution.
8. Close the Execution window.
9. Ensure that the Power On, ROM, and ROM Diagnostics steps are selected. Repeat steps 4 through 8, but select **Execute»Run Selected Steps Using»Single Pass** in step 4. TestStand executes the steps you selected using the Single Pass Execution entry point, which generates a UUT report.
10. Close the Execution window.

Running Selected Steps During an Execution

Complete the following steps to interactively execute selected steps in a sequence while suspended at a breakpoint during an execution.

1. Ensure that the Power On, ROM, and ROM Diagnostics steps are selected and select **Execute»Single Pass**.

2. Select the **ROM** test to fail and click **Done**. The execution stops at the breakpoint on the Power On step.
3. Step through the execution until you reach the RAM Diagnostics step. Notice that the ROM step failed.
4. Place a breakpoint at the ROM step in the Execution window and select the **ROM** and **ROM Diagnostics** steps.
5. Right-click the **ROM Diagnostics** step and select **Loop on Selected Steps** from the context menu to launch the Loop on Selected Steps dialog box. Enter **100** in the Loop Count control and click **OK**. TestStand starts an interactive execution of 100 loops for the steps you selected and enters a suspended state at the breakpoint for the ROM step. The execution pointer for the normal execution remains on the RAM Diagnostics step, and an execution pointer for the new interactive execution points to the ROM step.
6. Step through the interactive execution. The interactive execution toggles only between the ROM step and the ROM Diagnostics step. The ROM step continues to fail.
7. Click the **Terminate Interactive Execution** button on the Execution pane toolbar. TestStand returns the execution to a suspended state on the RAM Diagnostics step.
8. Complete the following steps to force the execution to continue from a step other than the currently suspended step.
 - a. Select the **ROM** step so it is the only highlighted step.
 - b. Right-click the **ROM** step and select **Set Next Step to Cursor** from the context menu. The execution pointer moves from the RAM Diagnostics step to the ROM step. The execution continues from the ROM step when you resume or step through the sequence.
 - c. Step over once. The ROM step executes instead of the RAM Diagnostics step.
9. Resume and complete the execution. The report contains entries for each step you executed interactively.
10. Remove the breakpoint from the Power On step in the Sequence File window.
11. Close all the windows in the sequence editor and do not save any changes.

Calling Sequences Dynamically

You can add to a sequence a step that dynamically runs one of two sequences, and you can pass parameters to the sequence you call.



Note Completed solution files are located in the `<TestStand Public>\Tutorial\Solution` directory.

Dynamically Specifying a Sequence to Run

Complete the following steps to open an existing sequence, add steps to prompt the operator for a CPU type and the number of CPUs to test, and add a step to call one of two different sequences depending on the type of CPU the user specifies.

1. Open `<TestStand Public>\Tutorial\Computer.seq`.
2. Select **File»Save <filename> As** and save the sequence as `Computer6.seq` in the `<TestStand Public>\Tutorial` directory.
3. Click the **Variables** pane in the Sequence File window, right-click the **Locals** item, select **Insert Local»String** from the context menu, and name the local variable `CPUType`.
4. Click the **Steps** pane in the Sequence File window to show the steps in the `MainSequence`.
5. Insert a Message Popup step below the Power On step, rename the new step `Select CPU Type`, and complete the following steps to configure the Select CPU Type step.
 - a. On the Text and Buttons tab of the Step Settings pane, enter the following values in the corresponding controls:
 - **Message Expression**—"Please select the CPU type for the UUT."
 - **Button 1**—"INTEL CPU"
 - **Button 2**—"AMD CPU"
 - **Cancel Button**—None
 - b. On the Layout tab of the Step Settings pane, enable the **Make Modal** option in the Dialog Options section. Enabling this option prevents the sequence editor from hiding the Select CPU dialog box and prevents you from interacting with the sequence editor until you close the Select CPU dialog box.
 - c. On the Properties tab of the Step Settings pane, click **Expressions** to show the Expressions panel.
 - d. In the Post-Expression control, enter the following expression to assign the string value "AMD" or "INTEL" to the local variable depending on the button users click:

```
Locals.CPUType = ((Step.Result.ButtonHit == 2) ? "AMD" : "INTEL")
```

Click the **Check Expression for Errors** button to verify that the expression contains valid syntax.

6. On the Steps pane, insert a Message Popup step below the Select CPU Type step, rename the new step `Specify Number of CPUs`, and complete the following steps to configure the Specify Number of CPUs step.
 - a. On the Text and Buttons tab of the Step Settings pane, enter the following values in the corresponding controls:
 - **Message Expression**—`"Please select the number of CPUs installed for the UUT."`
 - **Button 1**—`"1"`
 - **Button 2**—`"2"`
 - **Button 3**—`"3"`
 - **Button 4**—`"4"`
 - **Cancel Button**—None
 - b. On the Layout tab of the Step Settings pane, enable the **Make Modal** option in the Dialog Options section.
7. Insert a Sequence Call step below the Specify Number of CPUs step, rename the step `CPU Test`, and complete the following steps to configure the CPU Test step.
 - a. On the Module tab of the Step Settings pane, enable the **Specify By Expression** option.
 - b. Enter `Locals.CPUType + "Processor.seq"` in the File Path or Reference control and `"MainSequence"` in the Sequence control.
 - c. Click the **Load Prototype** button to launch the Load Sequence Prototype dialog box, in which you can select the prototype for the Sequence Call step.
 - d. Click the **Browse** button in the Load Sequence Prototype dialog box and select `<TestStand Public>\Tutorial\AMDProcessor.seq`.
 - e. Click **OK** to close the Load Sequence Prototype dialog box. TestStand populates the Parameters Table on the Module tab with the parameter list for the sequence.
 - f. Click in the Value column of the CPUsInstalled parameter and enter the following expression:

```
RunState.Sequence.Main["Specify Number of CPUs"].Result.ButtonHit
```

Click the **Check Expression for Errors** button to verify that the expression contains valid syntax.

8. Save the changes. Leave the sequence file open for the next tutorial.

Running a Sequence Dynamically

Complete the following steps to run a sequence dynamically.

1. Place a breakpoint at the CPU Test step.
2. Select **Execute»Single Pass**.
3. Click **Done** in the Test Simulator dialog box.
4. Click the **INTEL CPU** button in the Select CPU Type dialog box and click the **2** button in the Specify Number of CPUs dialog box.
5. When execution suspends at the breakpoint on the CPU Test step, click the **Step Into** button on the Execution window toolbar to step into the `MainSequence` sequence in `INTELProcessor.seq`.
6. Click the **Call Stack** pane. The call stack list shows `INTELProcessor.seq` at the top of the sequence call stack.
7. Click the tab for the Variables pane of the Execution window and expand the **Parameters** item. The value of the `CPUsInstalled` parameter equals the value on the button you clicked in the Specify Number of CPUs dialog box. The `MainSequence` sequence in the `INTELProcessor.seq` sequence file also requires a `ModelName` parameter. The Sequence Call step you created did not specify the `ModelName` parameter, so the TestStand Engine initializes the parameter to the default name.
8. Click the tab for the Steps pane of the Execution window and click the **Resume** button on the Execution window toolbar. When the execution completes, review the report but do not close the Execution window.
9. Click the **Restart** button on the Execution window toolbar to restart the execution and click **Done** in the Test Simulator dialog box. The Execution window must be the active window to restart the execution.
10. Click the **AMD CPU** button in the Select CPU Type dialog box and click the **3** button in the Specify Number of CPUs dialog box.
11. When the execution suspends at the breakpoint on the CPU Test step, step into the `MainSequence` sequence in `AMDProcessor.seq`. The Call Stack pane lists `AMDProcessor.seq` at the top of the call stack.
12. Resume and complete the execution and review the report.
13. Close all the windows in the sequence editor.

Customizing Reports

You can customize report generation within TestStand in a variety of ways. You can also create a custom results format by creating a custom result processing plug-in.



Note Completed solution files are located in the `<TestStand Public>\Tutorial\Solution` directory.

Configuring Test Report Options

Complete the following steps to configure the test report options.

1. Open `<TestStand Public>\Tutorial\Computer.seq`.
2. Select **Configure»Result Processing** to launch the Result Processing dialog box.
3. Click the icon in the Options column for the built-in Report model plug-in to launch the Report Options dialog box and complete the following steps.
 - a. On the Contents tab, select **ATML 6.01 Standards Report Document** from the Report Format ring control.
 - b. Enable the **Include Step Results** option and confirm the following step result settings:
 - Enable the **Include Test Limits** option.
 - Enable the **Include Measurements** option.
 - Select **Insert Graph** from the Include Arrays ring control.
 - c. Click the **Edit Format** button located to the right of the Default Numeric Format control to launch the Numeric Format dialog box, in which you specify the format TestStand uses to display the value of a numeric variable or property. By default, TestStand configures the numeric format to report numbers with 13 digits of precision.
 - d. Change **Maximum Number of Significant Digits** to 2 and click **OK** to close the Numeric Format dialog box.
4. Click the **Report File Pathname** tab, which you use to specify the report file pathname. You can specify a fixed pathname to use for all report files or you can specify options the report generator uses to generate report file pathnames. Use the default values for the options on this tab and click **OK** to close the Report Options dialog box.
5. Click **OK** to close the Result Processing dialog box.
6. Select **Execute»Test UUTs**. Run through several iterations of the sequence,

selecting components other than the Video and CPU tests to fail.

7. Click **Stop** in the UUT Information dialog box to stop sequence execution. The test report contains failure chain information for UUTs that fail. The failure chain shows the step failure that caused the UUT to fail and shows the Sequence Call steps through which the execution reached the failing step. Each step name in the failure chain links to the section of the report that shows the result for the step.
8. Close the Execution window.
9. Select **Configure»Result Processing** to launch the Result Processing dialog box.
10. Click the icon in the Options column for the built-in Report model plug-in to launch the Report Options dialog box and complete the following steps.
 - a. Click the **Contents** tab and select **ASCII Text File** from the Report Format ring control.
 - b. Select **Exclude Passed/Done/Skipped** from the Result Filtering Expression ring control for TestStand to apply a filtering expression to determine what steps appear in the report. In this example, you configure TestStand to record only the results of the steps that do not pass or steps that complete without any status. TestStand also changes the value of the Include Arrays ring control to Insert Table because ASCII reports do not support graphs.
11. Click **OK** to close the Report Options dialog box, and click **OK** to close the Result Processing dialog box.
12. Repeat steps 6 through 8 and examine the text version of the test report. Close the Execution window when you finish reviewing the report.

Using External Report Viewers

You can view the test report in external applications more suited for showing and editing text, such as Microsoft Word or Microsoft Excel.

Complete the following steps to use an external report viewer to view the test report.

1. Select **Configure»External Viewers** to launch the Configure External Viewers dialog box.
2. Click the **Add** button.
3. Click the **Browse** button located to the right of the Viewer control and navigate to the application you want to use to view the report, such as Word, and click **OK**.



Note When you select the application you want to use and click **OK**,

TestStand might launch the File Not Found dialog box because TestStand cannot locate the file within the default TestStand search directories, which TestStand uses to resolve relative paths to code modules and other files or directories. Select the **Use an absolute path for the file you selected** option to ensure that TestStand can access the application you want to use to view the report, and click **OK** to close the File Not Found dialog box.

4. Select **txt** from the Format pull-down menu.
5. Enable the **Automatically Launch Viewer** option.
6. Click **OK** to close the Configure External Viewers dialog box.
7. Select **Execute»Test UUTs**. Run through several iterations of the sequence, selecting components other than the Video and CPU tests to fail.
8. Click **Stop** in the UUT Information dialog box to stop sequence execution. TestStand generates the text report and launches the external viewer application you configured in step 3 to show the test report.
9. Examine the test report and close the external report viewing application.
10. Close the Execution window.
11. Complete the following steps to change the report settings back to the default settings.
 - a. Select **Configure»Result Processing** to launch the Result Processing dialog box.
 - b. Enable the **Show More Options** option.
 - c. Click the **Reset to Defaults** button.
 - d. Click **OK** in the Reset Configuration prompt.
 - e. Click **OK** to close the Result Processing dialog box.
12. Complete the following steps to change the external viewer settings back to the default settings.
 - a. Launch the Configure External Viewers dialog box.
 - b. Select the external viewer application you configured in step 3 and click the **Delete** button.
 - c. Click **OK** to close the Configure External Viewers dialog box.
13. Close all the windows in the sequence editor.

Adding Additional Results to Reports

Use the Additional Results panel to add and configure additional results, which are

values TestStand adds to the result list of a step when the step executes. An additional result can be a module parameter or a custom additional result in which you specify the name and value of the result. You can configure TestStand to automatically include additional results when generating a report or when logging results to a database. The default TestStand report generator style sheets do not display additional results for skipped steps.

Complete the following steps to create a step that calls a DLL code module to return a numeric array and add the numeric array and other values to the report.

1. Select **File»New»Sequence File** to open a new sequence file.
2. Save the sequence file as `CustomReport.seq` in the `<TestStand Public>\Tutorial` directory.
3. Complete the following steps to create a local variable.
 - a. Click the **Variables** pane in the Sequence File window, right-click the **Locals** item and select **Insert Local»Array of»Number** from the context menu to launch the Array Bounds dialog box, in which you can modify the array bounds.
 - b. Enter 49 in the Upper Bounds control and click **OK**.
 - c. Rename the local variable `NumArray`.
4. Click the **C/C++ DLL** adapter icon, as shown in the following figure and located at the top of the Insertion Palette.
5. Insert an Action step in the Main step group and complete the following steps to configure the Action step.
 - a. On the Module tab of the Step Settings pane, click the **Browse** button located to the right of the Module control, navigate to `<TestStand Public>\Tutorial\NumericArray.dll`, and click **Open**.
 - b. Select **GetNumericArray** from the Function ring control.
 - c. In the Value Expression column of the **measurements** parameter in the Parameters Table, enter `Locals.NumArray` to copy the value from the numeric array output parameter to the `Locals.NumArray` variable when TestStand returns from calling the `GetNumericArray` function.
 - d. In the Value column of the **Dim 1 Size** property in the Parameter Details Table located to the right of the Parameters Table, enter `-1` to specify that TestStand passes all elements of the `Locals.NumArray` property to the code module.
6. On the Properties tab of the Step Settings pane, complete the following steps to add the numeric array and other values to the report.

- a. Click **Additional Results** to show the Additional Results panel.
 - b. Place a checkmark in the **measurements [Out]** parameter checkbox to log the output value of the parameter.
 - c. Click the **Add Custom Result** button to add a new empty row to the list of the results.
 - d. Enter "Time", including the quotation marks, in the Name column and enter `Time()` in the Value to Log column. The `Time` TestStand expression function returns the current time.
 - e. Click the **Add Custom Result** button again.
 - f. Enter "Date", including the quotation marks, in the Name column and enter `Date()` in the Value to Log column. The `Date` TestStand expression function returns the current date.
7. Save the changes you made and select **Execute»Single Pass**. Review the report that includes the array data, the time, and the date.
 8. Close all the windows in the sequence editor.

Adding to Reports Using Callbacks

Complete the following steps to add a logo to the header of the HTML report using a Report callback in the process model.

1. Open `<TestStand Public>\Tutorial\Computer.seq`.
2. Select **File»Save <filename> As** and save the sequence file as `Computer7.seq` in the `<TestStand Public>\Tutorial` directory.
3. Select **Configure»Result Processing** to launch the Result Processing dialog box.
4. Click the icon in the Options column for the built-in Report model plug-in to launch the Report Options dialog box and complete the following steps.
 - a. On the Contents tab, select **HTML Document** from the Report Format ring control.
 - b. Click **OK** to close the Report Options dialog box, and click **OK** to close the Result Processing dialog box.
5. Select **Edit»Sequence File Callbacks** to launch the Sequence File Callbacks dialog box, select the **ModifyReportHeader** callback, and click **Add** to add the callback to the sequence file.
6. Click **Edit** to close the Sequence File Callbacks dialog box and edit the new **ModifyReportHeader** callback sequence in the Sequence File window, which TestStand automatically opens.
7. Click the **Variables** pane, right-click the **Locals** item, select **Insert Local»String**

from the context menu, and name the local variable `AddToHeader`. Click in the Value column of the `AddToHeader` variable and enter the following text:

```
<img alt='Logo Goes Here' src='Logo.jpg'><br><br> <a  
href='http://www.ni.com'>Visit Our Website</a><br><br>
```

8. Insert a Statement step in the Main step group and rename the step `Add Custom Logo`.
9. On the Expression tab of the Step Settings pane, enter the following expression in the Expression control:

```
Parameters.ReportHeader = Locals.AddToHeader + Parameters.ReportHeader
```

Click the **Check Expression for Errors** button to verify that the expression contains valid syntax.

10. Save the changes and select **Execute»Single Pass**.
11. Click **Done** in the Test Simulator dialog box. Review the report after the execution completes and notice the new logo image at the top of the UUT report.
12. Complete the following steps to change the report settings back to the default settings.
 - a. Select **Configure»Result Processing** to launch the Result Processing dialog box.
 - b. Enable the **Show More Options** option.
 - c. Click the **Reset to Defaults** button.
 - d. Click **OK** in the Reset Configuration prompt.
 - e. Click **OK** to close the Result Processing dialog box.
13. Close all the windows in the sequence editor.

Calling LabVIEW VIs

Use the LabVIEW Adapter to call VIs from TestStand, to create new LabVIEW projects and VIs to call from TestStand, to edit and debug existing VIs, and to use LabVIEW data types in TestStand.



Note Completed solution files are located in the `<TestStand Public>\Tutorial\Solution` directory.

Required LabVIEW Settings

All of the tutorials on this page require you to have the LabVIEW development system and TestStand installed on the same computer. In addition, you must configure the LabVIEW Adapter to run VIs using the LabVIEW development system.

Confirm the following settings in LabVIEW:

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

Creating and Configuring Steps with the LabVIEW Adapter

You can create steps that call stand-alone VIs, call VIs in the context of a LabVIEW project, and call LabVIEW class member VIs.

Calling a Stand-alone VI

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

1. Select **File»New»Sequence File** to open a new sequence file.
2. Save the sequence file as `Call LabVIEW VI.seq` in the `<TestStand Public>\Tutorial` directory.
3. Select the **LabVIEW** adapter from the Adapter ring control above the Step Types list.
4. Insert a Pass/Fail Test step in the Main step group and rename the new step `LabVIEW Pass/Fail Test`.
5. On the LabVIEW Module tab of the Step Settings pane, click the **Browse for VI** button located to the right of the VI Path control, navigate to `<TestStand Public>\Tutorial\LabVIEW Pass-Fail Test.vi`, and click **Open**. TestStand reads the description and connector pane information from the VI and updates the LabVIEW Module tab so you can configure the data to pass to and from the VI.

6. In the VI Parameter Table, enter the values shown in Table 11-1.

Parameter Name	Value
PASS/FAIL Flag	<code>Step.Result.PassFail</code>
Report Text	<code>Step.Result.ReportText</code>

When TestStand calls the VI, it places the value the VI returns in the PASS/FAIL Flag and Report Text indicators into the `Result.PassFail` and `Result.ReportText` properties of the step, respectively. Notice that TestStand automatically fills in the Value column of the **error out** output parameter with the `Step.Result.Error` property. By default, when a VI uses the standard LabVIEW **error out** cluster as an output parameter, TestStand automatically passes that value into the `Step.Result.Error` property for the step. You can also update the value manually. If an error occurs during execution of the VI and the **error out** cluster is passed to `Step.Result.Error`, TestStand launches the Run-Time Error dialog box by default.

7. Save the changes. Leave the sequence file open for the next tutorial.

Calling a VI in the Context of a LabVIEW Project

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

1. Insert another Pass/Fail Test step in the Main step group and rename the new step `LabVIEW Project Pass/Fail Test`.
2. On the LabVIEW Module tab, click the **Browse for LabVIEW Project** button located to the right of the Project Path control, navigate to `<TestStand Public>\Tutorial\Test Project.lvproj`, and click **Open**. Notice that some of the TestStand button icons and tool tips change to indicate support for LabVIEW projects.
3. Click the **Browse for VI in LabVIEW Project** button, located to the right of the VI Path control, to launch the Select VI from LabVIEW Project dialog box.
4. Select `LabVIEW ProjectPass-Fail Test.vi` and click **OK**. Notice that the path defined in the project is selected as the VI path.
5. In the VI Parameter Table, enter the values shown in Table 11-2.

Parameter Name	Value
PASS/FAIL Flag	<code>Step.Result.PassFail</code>
Report Text	<code>Step.Result.ReportText</code>

6. Save the changes. Leave the sequence file open for the next tutorial.

Calling LabVIEW Class Member VIs

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



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

Calling a Static Member VI from a LabVIEW Class

Complete the following steps to insert a new LabVIEW step and configure the step to call a static member VI from a LabVIEW class in order to create a LabVIEW class object.

1. Insert an Action step in the Main step group and rename the new step `Create LabVIEW Class Object`.
2. On the LabVIEW Module tab, select **Class Member Call** from the Call Type ring control. Notice that some of the TestStand button icons and tool tips change to indicate support for LabVIEW class member calls.
3. Click the **Browse for LabVIEW Project** button, navigate to `<TestStand Public>\Tutorial\Test Project.lvproj`, and click **Open**.
4. Click the **Browse for LabVIEW Class in LabVIEW Project** button, located to the right of the Class Path control, to launch the Select a Class from LabVIEW Project dialog box.
5. Select `LabVIEW Child Class.lvclass` and click **OK**.
6. Select **LabVIEW Child Static Factory Member.vi** from the Member Name ring control.
7. In the VI Parameter Table, enter `Locals.myChildObject` in the Value column

of the **LabVIEW Child Class out** output parameter. The `myChildObject` local variable name appears in red text because it does not exist yet. You create this variable in the next step.

8. Right-click the `Locals.myChildObject` value and select **Create Locals.myChildObject»Object Reference** from the context menu to create `Locals.myChildObject` as an object reference local variable, which appears on the Variables pane.
9. Save the changes. Leave the sequence file open for the next tutorial.

Calling a Dynamically Dispatched Member Method VI

Complete the following steps to insert a new LabVIEW step and configure the step to call a dynamically dispatched member method VI on the LabVIEW class object that the Create LabVIEW Class Object step creates.

1. Insert another Pass/Fail Test step in the Main step group and rename the new step `LabVIEW Class Member Pass/Fail Test`.
2. On the LabVIEW Module tab, select **Class Member Call** from the Call Type ring control.
3. Click the **Browse for LabVIEW Project** button, navigate to `<TestStand Public>\Tutorial\Test Project.lvproj`, and click **Open**.
4. Click the **Browse for LabVIEW Class in LabVIEW Project** button to launch the Select a Class from LabVIEW Project dialog box.
5. Select **LabVIEW Parent Class.lvclass** and click OK.
6. Select **LabVIEW Dynamic Pass-Fail Test Member.vi** from the Member Name ring control.
7. In the VI Parameter table, enter the values shown in Table 11-3.

Parameter Name	Value
LabVIEW Parent Class in	<code>Locals.myChildObject</code>
LabVIEW Parent Class out	<code>Locals.myChildObject</code>
PASS/FAIL Flag	<code>Step.Result.PassFail</code>
Report Text	<code>Step.Result.ReportText</code>

8. Save the changes. Leave the sequence file open for the next tutorial.

Executing the Sequence

Complete the following steps to execute the sequence of steps.

1. Select **Execute»Single Pass**. When the execution completes, the report indicates the steps passed.
2. Close the Execution window. Leave the sequence file open for the next tutorial.



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

Creating and Editing VIs

You can use the LabVIEW Adapter to create a new VI from TestStand, create a new LabVIEW project and add a new VI to the project from TestStand, and edit an existing VI from TestStand.

Creating a New VI from TestStand

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

1. Select **File»Save <filename> As** and save the sequence file as `Call LabVIEW VI 2.seq` in the `<TestStand Public>\Tutorial` directory.
2. Insert a Numeric Limit Test step after the LabVIEW Class Member Pass/Fail Test step and rename the new step `LabVIEW Numeric Limit Test`.
3. On the LabVIEW Module tab, complete the following steps to configure the LabVIEW Numeric Limit Test step.
 - a. Click the **Create VI** button, located to the right of the VI Path control, to create a new VI.
 - b. In the File dialog box, browse to the `<TestStand Public>\Tutorial` directory, enter `LabVIEW Numeric Limit Test.vi` in the File Name control, and click **OK**. TestStand creates a new VI based on the available code templates for the TestStand Numeric Limit Test and opens the VI in LabVIEW.



Note The TestStand Numeric Limit Test step type requires code modules to store a measurement value in the

`Step.Result.Numeric` property, and the step type performs a comparison operation to determine whether the step passes or fails. Code modules can pass step properties as parameters to and from the code module or use the TestStand API in the code module to update step properties. When you use a default code template from NI to create a code module, TestStand creates the parameters needed to access the step properties for you.

- c. In LabVIEW, select **Window»Show Block Diagram** to open the block diagram.
- d. Right-click the **Numeric Measurement** indicator terminal, select **Create»Constant** from the context menu, and enter `10.0`.
- e. Save and close the VI.
4. In TestStand, click the **LabVIEW Module** tab. Notice that TestStand automatically updates the output parameters for the VI based on the information stored in the code template for the Numeric Limit Test step type.
5. Save the changes and select **Execute»Single Pass**. When the execution completes, the report indicates the step passed with a numeric measurement of `10`.
6. Close the Execution window. Leave the sequence file open for the next tutorial.

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

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

1. Insert another Numeric Limit Test step after the LabVIEW Numeric Limit Test step and rename the new step `LabVIEW Project Numeric Limit Test`.
2. On the LabVIEW Module tab, complete the following steps to configure the LabVIEW Project Numeric Limit Test step.
 - a. Click the **Create LabVIEW Project** button, located to the right of the Project Path control, to create a new LabVIEW Project.
 - b. In the File dialog box, browse to the `<TestStand Public>\Tutorial` directory, enter `LabVIEW Project Test.lvproj` in the File Name control, and click **OK**. TestStand creates a new LabVIEW Project and opens the project in LabVIEW.
 - c. In TestStand, click the **Add or Remove VIs from LabVIEW Project** button, located to the right of the VI Path control, to launch the Add or Remove Items from LabVIEW Project dialog box.

- d. Right-click the **My Computer** item in the LabVIEW Project control and select **New VI** from the context menu to launch the Select the LabVIEW VI to Create dialog box.
 - e. In the Select LabVIEW VI to Create dialog box, browse to the <TestStand Public>\Tutorial directory, enter LabVIEW Project Numeric Limit Test.vi in the File Name control, and click **OK**. TestStand creates a new VI based on the available code templates for the TestStand Numeric Limit Test step type and adds the VI to the LabVIEW project.
 - f. Right-click the new VI under the My Computer item and select **Edit VI** from the context menu. TestStand opens the VI in LabVIEW within the project context, which LabVIEW displays in the bottom left corner of the front panel.
 - g. In LabVIEW, open the block diagram for the VI.
 - h. Right-click the **Numeric Measurement** indicator terminal, select **Create»Constant** from the context menu, and enter 10.0.
 - i. Save and close the VI.
3. In TestStand, select LabVIEW Project Numeric Limit Test.vi in the Add or Remove Items from the LabVIEW Project dialog box, and click **Select** to close the Add or Remove Items from LabVIEW Project dialog box. On the LabVIEW Module tab, notice that TestStand automatically updates the output parameters for the VI based on the information stored in the code template for the Numeric Limit step type.
 4. Save the changes and select **Execute»Single Pass**. When the execution completes, the report indicates the steps passed with a numeric measurement of 10.0.
 5. Close the Execution window. Leave the sequence file open for the next tutorial.

Editing Existing VIs from TestStand

Complete the following steps to edit existing VIs from TestStand.

1. Select the **LabVIEW Pass/Fail Test** step and use the LabVIEW Module tab to complete the following steps.
 - a. Click the **Edit VI** button, located to the right of the VI Path control. LabVIEW becomes the active application in which the LabVIEW Pass-Fail Test VI is open.
 - b. Open the block diagram for the VI and change the **PASS/FAIL Flag** Boolean constant to `False`.
 - c. Save and close the VI.
2. In TestStand, select the **LabVIEW Project Pass/Fail Test** step and use the LabVIEW

Module tab to complete the following steps.

- a. Click the **Edit VI** button. LabVIEW becomes the active application and the LabVIEW Project Pass-Fail Test VI is open within the LabVIEW project context.
- b. Open the block diagram for the VI and change the **PASS/FAIL Flag** Boolean constant to `False`.
- c. Save and close the VI.
3. In TestStand, save the changes and select **Execute»Single Pass**. When the execution completes, the report indicates the LabVIEW Pass/Fail Test and LabVIEW Project Pass/Fail Test steps failed.
4. Close the Execution window. Leave the sequence file open for the next tutorial.

Debugging VIs

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

1. Place a breakpoint at the LabVIEW Pass/Fail Test step.
2. Select **Execute»Run MainSequence**. The execution pauses at the LabVIEW Pass/Fail Test step.
3. Complete the following steps to debug the LabVIEW Pass-Fail Test VI the LabVIEW Pass/Fail Test step calls.
 - a. Click the **Step Into** button at the top of the Execution window. LabVIEW becomes the active application, in which the LabVIEW Pass-Fail Test VI is open and in a suspended state.
 - b. Open the block diagram of the suspended VI.
 - c. Click the **Step Into** or **Step Over** button on the LabVIEW toolbar to begin stepping through the VI. You can click the **Finish VI** button at any time to finish stepping through the VI.
 - d. When you finish stepping through the VI, click the **Return to Caller** button on the LabVIEW toolbar to return to TestStand. The execution pauses at the next step in the sequence.
4. Click the **Resume** button on the Debug toolbar in TestStand to complete the execution.
5. Close the Execution window.
6. Remove the breakpoint from the LabVIEW Pass/Fail Test step. Leave the sequence file open for the next tutorial.

Creating TestStand Data Types from LabVIEW Clusters

TestStand provides number, string, Boolean, and object reference built-in data types. TestStand also provides several standard named data types, including Path, Error, LabVIEWAnalogWaveform, and others. You can create container data types to hold any number of other data types. TestStand container data types are analogous to LabVIEW clusters.

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

1. Select **File»Save <filename> As** and save the sequence file as `Call LabVIEW VI 3.seq` in the `<TestStand Public>\Tutorial` directory.
2. Insert a Pass/Fail Test step in the Main step group after the LabVIEW Project Numeric Limit Test step and rename the new step `Pass Container` to `VI`.
3. On the LabVIEW Module tab, click the **Browse for VI** button, navigate to `<TestStand Public>\Tutorial\VI with Cluster Input.vi`, and click **Open**.
4. Click the **Create/Update Custom Data Type** button, located in the Type column of the **Input Cluster** parameter in the VI Parameter Table, to launch the Create/Update Custom Data Type From Cluster dialog box. TestStand maps the cluster elements to subproperties in a container called `Input_Cluster`, which is a new TestStand custom data type. You can rename the data type and subproperties as necessary and specify where TestStand stores the new data type.
5. In the Create/Update Custom Data Type From Cluster dialog box, change the type name to `InputData` and click the **Create** button to accept the automatically assigned values and to create the data type in the current sequence file.
6. On the LabVIEW Module tab, remove the checkmark from the Default column for the **Input Cluster** input parameter, click the **Expression Browse** button in the Value column to launch the Expression Browser dialog box, and complete the following steps.
 - a. On the Variables/Properties tab, right-click the **Locals** item and select **Insert Types»InputData** to create a local variable of the `InputData` data type. Rename the local variable `ContainerData`.
 - b. Right-click the `Number` subproperty of `ContainerData` and select **Properties** from the context menu to launch the Number Properties dialog box.

- c. Enter 23 in the Value control and click **OK**.
 - d. Right-click the `String` subproperty of `ContainerData` and select **Properties** from the context menu to launch the String Properties dialog box.
 - e. Enter `My String Data` in the Value control and click **OK**.
 - f. Enter `Locals.ContainerData` in the **Expression** control on the Variables/Properties tab and click **OK**. The Value column for the **Input Cluster** parameter now contains `Locals.ContainerData`.
7. Enter `Step.Result.ReportText` in the **Value** column for the **Report Text** output parameter. When TestStand calls the VI, it passes the values in the `ContainerData` local variable to the **Input Cluster** control on the VI and returns the **Number** and **String** elements of the **Input Cluster** parameter to the `ReportText` property of the step.
 8. Save the changes and select **Execute»Single Pass**. When the execution completes, the report shows the text the VI with Cluster Input VI returns.
 9. Close all the windows in the sequence editor.

Calling LabWindows/CVI Code Modules

Use the LabWindows/CVI Adapter to call LabWindows/CVI code modules from TestStand, to create new code modules to call from TestStand, to edit and debug existing code modules, and to use LabWindows/CVI data types in TestStand.



Note Completed solution files are located in the `<TestStand Public>\Tutorial\Solution` directory.

Required LabWindows/CVI Settings

All of the tutorials on this page require you to have the LabWindows/CVI development system and TestStand installed on the same computer.

In addition, complete the following steps to configure the LabWindows/CVI Adapter to execute steps in an external instance of the LabWindows/CVI development system and to allow only new code templates.

1. Select **Configure»Adapters** to launch the Adapter Configuration dialog box.
2. Select **LabWindows/CVI** in the Adapter column and click the **Configure** button to launch the LabWindows/CVI Adapter Configuration dialog box.

3. In the Step Execution section, select the **Execute Steps in an External Instance of CVI** option.
4. In the Code Template Policy section, select the **Allow Only New Templates** option.
5. Click **OK** to close the LabWindows/CVI Adapter Configuration dialog box, click **OK** again when the adapter displays a warning that confirms TestStand will unload all modules, and click **Done** in the Adapter Configuration dialog box.

Creating and Configuring Steps with the LabWindows/CVI Adapter

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

1. Select **File»New»Sequence File** to open a new sequence file.
2. Save the sequence file as `CallCVICodeModule.seq` in the `<TestStand Public>\Tutorial` directory.
3. Select the **LabWindows/CVI** adapter from the Adapter ring control above the Step Types list.
4. Insert a Pass/Fail Test step in the Main step group and rename the new step `CVI Pass/Fail Test`.
5. On the LabWindows/CVI Module tab of the Step Settings pane, click the **File Browse** button located to the right of the Module control, navigate to `<TestStand Public>\Tutorial\CallCVICodeModule.dll`, and click **Open**.
6. Select **PassFailTest** from the Function ring control.



Note When you select a function, the LabWindows/CVI Adapter attempts to read the export information LabWindows/CVI includes in the DLL or the function parameter information from the type library in the code module, if one exists. When the function parameter information is not defined, you can select a code template from the Code Template ring control to specify the function prototype or add parameters to the Parameters Table control to specify the function prototype.

7. Select **PassFail template for LabWindows/CVI** from the Code Template ring control. The Parameters Table control contains the default value expressions the code template specifies. When TestStand calls the code module, the LabWindows/CVI Adapter stores the returned values for the result and the error details in the specified properties of the step.

8. Save the changes and select **Execute»Single Pass**. Because the LabWindows/CVI Adapter is configured to use an external instance of LabWindows/CVI to execute code modules, TestStand launches the LabWindows/CVI development environment to execute the function the step calls. When the execution completes, the report indicates the step passed.
9. Select **File»Unload All Modules** to unload the DLL the step calls so you can rebuild the DLL in a subsequent tutorial.
10. Close the Execution window. Leave the sequence file open for the next tutorial.

Creating a New Code Module from TestStand



Note NI recommends using DLL files when you develop code modules using the LabWindows/CVI Adapter. The following tutorials demonstrate creating and debugging only DLL code modules.

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

1. Select **File»Save <filename> As** and save the sequence file as `CallCVIcodeModule2.seq` in the `<TestStand Public>\Tutorial` directory.
2. Insert a Numeric Limit Test step after the CVI Pass/Fail Test step and rename the new step `CVI Numeric Limit Test`.
3. On the LabWindows/CVI Module tab, complete the following steps to configure the CVI Numeric Limit Test step.
 - a. Click the **File Browse** button, navigate to `<TestStand Public>\Tutorial\CallCVIcodeModule.dll`, and click **Open**.
 - b. Enter `NumericLimitTest` in the Function ring control.
 - c. Select **NumericLimit template for LabWindows/CVI** from the Code Template ring control. Notice that TestStand automatically updates the function prototype and parameter values based on the information stored in the code template for the Numeric Limit Test step type.
4. Click the **Source Code Files** button, located to the right of the Parameter Details Table control, to launch the CVI Source Code Files window and complete the following steps.
 - a. Enter `CVINumericLimitTest.c` in the Source File Containing Function control.
 - b. For the CVI Project File to Open control, click the **File Browse** button, navigate

to `<TestStand Public>\Tutorial\CallCVIcodeModule.prj`, and click **Open**.

c. Click **Close**.

5. Click the **Create Code** button, located just below the Source Code Files button, to create a code module. When you click the **Create Code** button, TestStand launches the Select a Source File dialog box, in which you can specify the source code to use for the code module.
6. Navigate to the `<TestStand Public>\Tutorial` directory and click **OK**. TestStand creates a new code module source file named `CVINumericLimitTest.c` based on the available code templates for the TestStand Numeric Limit Test step type and opens the code module in LabWindows/CVI.



Note The TestStand Numeric Limit Test step type requires code modules to store a measurement value in the `Step.Result.Numeric` property, and the step type performs a comparison operation to determine whether the step passes or fails. Code modules can pass step properties as parameters to and from the code module or use the TestStand API in the code module to update step properties. When you use a default code template from NI to create a code module, TestStand creates the parameters needed to access the step properties for you.

7. In LabWindows/CVI, uncomment the following code in the source file: `double testMeasurement = 10.0; double lowLimit; *measurement = testMeasurement;`
8. Save and close the source file. Leave LabWindows/CVI open.
9. In the LabWindows/CVI project window, select **Build»Create Debuggable Dynamic Link Library** to rebuild the DLL. Click **OK** when LabWindows/CVI prompts you that the files are created.
10. In TestStand, save the changes and select **Execute»Single Pass**. When the execution completes, the report indicates the step passed with a numeric measurement of 10.
11. Select **File»Unload All Modules** to unload the DLL.
12. Close the Execution window. Leave the sequence file and LabWindows/CVI open for the next tutorial.

Editing an Existing Code Module from TestStand

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

1. Select the **CVI Numeric Limit Test** step and use the LabWindows/CVI Module tab to complete the following steps.
 - a. Click the **Edit Code** button, located just below the **Create Code** button. LabWindows/CVI becomes the active application in which the `CVINumericLimitTest.c` source file is open.
 - b. Change the initial value in the declaration for the testMeasurement variable to `5.0`.
 - c. Save and close the source file. Leave LabWindows/CVI open.
2. Rebuild the DLL. Click **OK** when LabWindows/CVI prompts you that the files are created.
3. In TestStand, select **Execute»Single Pass**. When the execution completes, the report indicates the step failed with a numeric measurement of 5.
4. Close the Execution window. Leave the sequence file and LabWindows/CVI open for the next tutorial.

Debugging a Code Module

Complete the following steps to debug a code module you call from TestStand using the LabWindows/CVI Adapter.

1. Place a breakpoint at the CVI Pass/Fail Test step.
2. Select **Execute»Run MainSequence**. The execution pauses at the CVI Pass/Fail Test step.
3. Complete the following steps to debug the code module the CVI Pass/Fail Test step calls.
 - a. Click the **Step Into** button on the Debug toolbar in TestStand. LabWindows/CVI becomes the active application, in which the LabWindows/CVI Pass-Fail Test code module is open and in a suspended state.
 - b. Click the **Step Into** button or the **Step Over** button on the LabWindows/CVI toolbar to begin stepping through the code module.
 - c. When you finish stepping through the code module, click the **Finish Function** button on the LabWindows/CVI toolbar to return to TestStand. The execution pauses at the next step in the sequence.
4. Click the **Resume** button on the Debug toolbar in TestStand to complete the

execution.

5. Close the Execution window.
6. Select **File»Unload All Modules** to unload the DLL.
7. Remove the breakpoint from the CVI Pass/Fail Test step. Leave the sequence file and LabWindows/CVI open for the next tutorial.

Creating TestStand Data Types from LabWindows/CVI Structs

TestStand provides number, string, Boolean, and object reference built-in data types. TestStand also provides several standard named data types, including Path and Error. You can create container data types to hold any number of other data types. TestStand container data types are analogous to C structures in LabWindows/CVI.

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

Creating a New Custom Data Type

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

1. Click the **Types** button on the sequence editor toolbar to open the Types window.
2. Select the `CallCVICodeModule2.seq` sequence file on the View Types For pane.
3. Right-click the **Custom Data Types** item and select **Insert Custom Data Type»Container** from the context menu to insert a new data type. Rename the new container data type `CVITutorialStruct`.
4. Expand the `CVITutorialStruct` item.
5. Right-click the `CVITutorialStruct` item and select **Insert Field»Number** from the context menu to insert a new field in the data type. Rename the new field `Measurement`.
6. Right-click the `CVITutorialStruct` item and select **Insert Field»String** from the context menu to insert another new field in the `CVITutorialStruct` container data type. Rename the new field `Buffer`.
7. Save the changes. If TestStand warns you that the sequence file contains modified types, review the information in the warning dialog box, select the **Increment Type Versions** option, and click **OK**. Leave the sequence file open for the next tutorial.

Specifying Structure Passing Settings

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

1. Right-click the `CVITutorialStruct` item in the Types window and select **Properties** from the context menu to launch the Type Properties dialog box. The name of the Type Properties dialog box is specific to the name of the property you select.
2. Click the **C Struct Passing** tab of the Type Properties dialog box.
3. Enable the **Allow Objects of This Type to be Passed as Structs** option. The Property ring control lists the two fields in the `CVITutorialStruct` container data type. Notice that the Numeric Type control for the Measurement property defaults to 64-bit Real Number (double).
4. Select the **Buffer** property from the Property ring control.
5. For the String Type control, select the **C String Buffer** option to allow the C function to alter the value of the structure field.
6. Click the **Version** tab of the Type Properties dialog box and disable the **Modified** option.
7. Select **OK** to close the Type Properties dialog box.
8. Save the changes. Close the Types window. Leave the sequence file open for the next tutorial.

Calling a Function with a Struct Parameter

Complete the following steps to use the `CVITutorialStruct` container data type as a parameter to a function a step calls.

1. Select **File»Save <filename> As** and save the sequence file as `CallCVIModule3.seq` in the `<TestStand Public>\Tutorial` directory.
2. Click the **Variables** pane.
3. Right-click the **Locals** item, select **Insert Local»Type»CVITutorialStruct** from the context menu to insert an instance of the container data type, and rename the new variable `CVIStruct`.
4. Insert an Action step into the Main step group after the CVI Numeric Limit Test step and rename the new step `Pass Struct Test`.
5. On the LabWindows/CVI Module tab, click the **File Browse** button, navigate to `<TestStand Public>\Tutorial\CallCVICodeModule.dll`, and click

Open.

6. Enter `PassStructTest` in the Function control.
7. Click the **Add Parameter** button, located to the right of the Parameters Table control, to insert a new parameter in the Parameters Table control.
8. Enter the following information in the Parameter Details Table control:
 - a. In the **Name** field, rename the parameter `cviStruct`.
 - b. In the **Category** field, select `C Struct`.
 - c. In the **Type** field, select `CVITutorialStruct`.
9. Enter `Locals.CVIStruct` in the Value Expression column for the parameter in the Parameters Table control.
10. Click the **Source Code Files** button to launch the CVI Source Code Files window. Complete the following steps to select the source file and project file to use when inserting the function.
 - a. Enter `CVIStructPassingTest.c` in the Source File Containing Function control.
 - b. For the CVI Project File to Open control, click the **File Browse** button, navigate to `<TestStand Public>\Tutorial\CallCVICodeModule.prj`, and click **Open**.
 - c. Click **Close**.
11. Click the **Create Code** button to create a code module. TestStand launches the Select a Source File dialog box.
12. Browse to the `<TestStand Public>\Tutorial` directory and click **OK**. TestStand creates a new source file named `CVIStructPassingTest.c` with an empty function.
13. In LabWindows/CVI, complete the following steps.
 - a. Add the following type definition before the `PassStructTest` function:

```
struct CVITutorialStruct { double measurement; char buffer[256]; };
```

- b. Add the following code to the `PassStructTest` function:

```
if (cviStruct) { cviStruct->measurement = 10.0; strcpy(cviStruct->buffer, "Average Voltage"); }
```

- c. Add the following statement to the top of the source file to include the declaration of the `strcpy` function:

```
#include <ansi_c.h>
```

14. Save and close the source file.

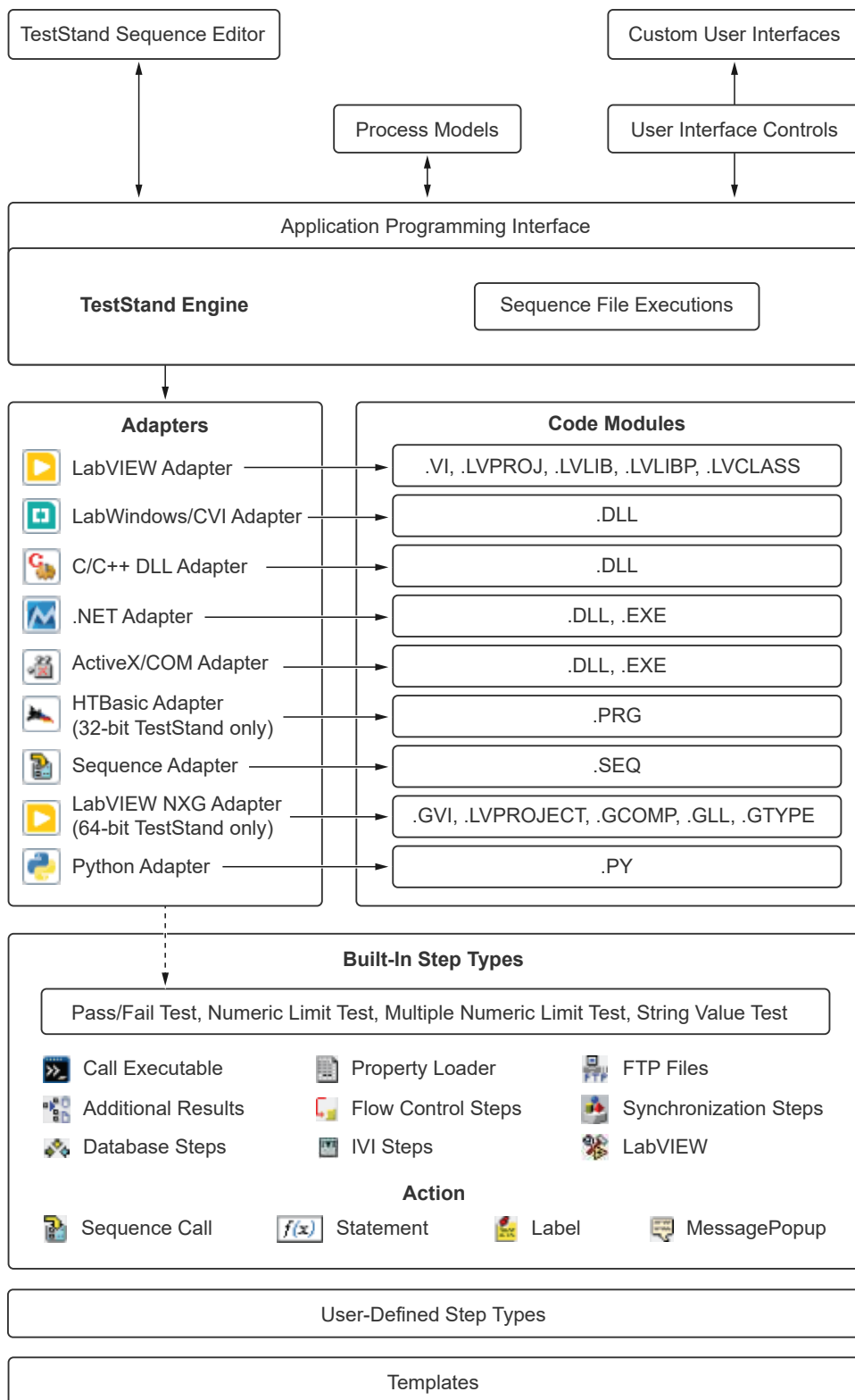
15. In the LabWindows/CVI project window, select **Build»Create Debuggable Dynamic Link Library** to rebuild the DLL. Click **OK** when LabWindows/CVI notifies you that the files are created.
16. In TestStand, place a breakpoint at the new Pass Struct Test step.
17. Save the changes and select **Execute»Run MainSequence**.
18. Step through the sequence and review the values in the **Locals.CVIStructvariable** on the Variables pane of the Execution window before and after executing the Pass Struct Test step.
19. Select **File»Unload All Modules** to unload the DLL.
20. Remove the breakpoint from the CVI Pass/Fail Test step. Close all the windows in the sequence editor.

Related concepts:

- [Calling LabVIEW VIs](#)

Parts of a TestStand Application

The following figure and corresponding descriptions illustrate how the major parts of a TestStand application interact.



- **TestStand Sequence Editor**—The TestStand Sequence Editor is a development environment in which you create, edit, execute, and debug sequences and the tests sequences call. Use the sequence editor to create and edit other components,

such as step types and process models.

- **TestStand User Interfaces**—A TestStand User Interface is an application you deploy to a development system or a production station to provide a custom GUI for executing, debugging, or editing sequences. Simple user interfaces might only support running sequences, while more complex user interfaces or custom sequence editors might support editing, running, and debugging sequences.
- **TestStand User Interface Controls**—The user interfaces use the TestStand User Interface (UI) Controls, a collection of ActiveX controls for creating custom user interfaces and sequence editors in TestStand. These controls simplify common user interface tasks, such as displaying sequences and executions. You can use these controls in any programming environment that can host ActiveX controls.
- **TestStand Engine**—The TestStand Engine is a set of DLLs that exports an ActiveX Automation API. The TestStand Sequence Editor and UI Controls use the TestStand API, which you can call from any programming environment that supports access to ActiveX servers, including code modules you write in LabVIEW and LabWindows/CVI.
- **Module Adapters**—The TestStand Engine uses module adapters to invoke code modules written in a variety of languages and application development environments (ADEs), including LabVIEW and LabWindows/CVI. Module adapters load and call code modules, pass parameters to code modules, and return values and status from code modules. TestStand handles core test management functionality, including the definition and execution of the overall testing process, user management, report generation, database logging, and more.
- **TestStand Deployment Utility**—After you develop, customize, and debug a TestStand system, you can deploy the system to multiple test stations. The TestStand Deployment Utility simplifies the complex process of deploying a TestStand system by automating many of the steps involved, including collecting sequence files, code modules, configuration data for instruments, and support files for the test system. You can also use the deployment utility to create a full deployment installer or a patch deployment installer.

Related concepts:

- [Step Types](#)
- [Process Models](#)
- [Creating Custom User Interfaces](#)
- [TestStand UI Controls](#)
- [TestStand ActiveX API Overview](#)

- [Module Adapters](#)
- [Building a Customized MSI-based Installer](#)
- [Patching Deployments](#)
- [Deploying TestStand Systems](#)

General Test Executive Concepts

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

The following concepts are applicable to test executives in general:

- **Code module**—A program module, such as a Microsoft Windows DLL or LabVIEW VI (.vi), that contains one or more functions that perform a specific test or other action.
- **Step**—An individual element of a test sequence that can call code modules or perform other operations.
- **Sequence**—A series of steps you specify to execute in a particular order. Whether and when a step executes depends on the results of previous steps.
- **Subsequence**—A sequence another sequence calls as a step.
- **Sequence file**—A file that contains the definition of one or more sequences.
- **Sequence editor**—A program that provides a graphical user interface (GUI) for creating, editing, executing, and debugging sequences.
- **User interface**—A program that provides a GUI for executing sequences on a production station. A sequence editor and user interface can be separate applications or different aspects of the same application.
- **Test executive engine**—A module or set of modules that provide an API for creating, editing, executing, and debugging sequences. A sequence editor or user interface uses the services of a test executive engine.
- **Application Development Environment (ADE)**—A programming environment, such as LabVIEW, LabWindows™/CVI™, or Microsoft Visual Studio, in which you create code modules and user interfaces.
- **Unit Under Test (UUT)**—The device or component to test.
- **Deployment**—An image or an installer for an image to transfer a test system from development to production.

TestStand Building Blocks

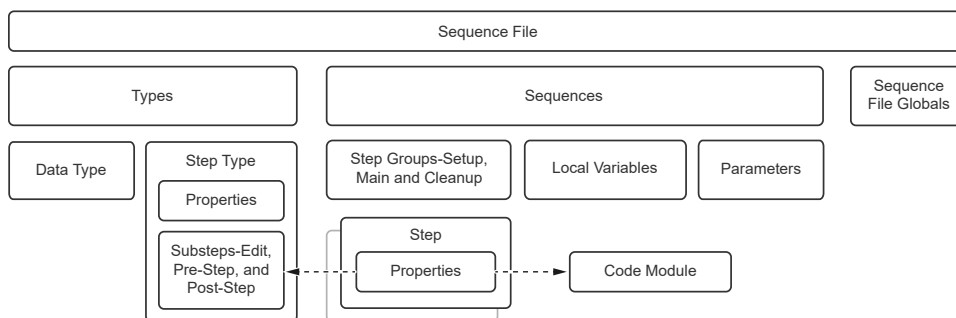
Use the following TestStand building blocks to create test systems.

Sequences

A sequence contains a series of steps and can consist of the following components:

- Setup step group
- Main step group
- Cleanup step group
- Sequence local variables
- Sequence file global variables
- Sequence parameters
- Callback sequences

The following figure illustrates the structure of a sequence file.



Built-In Sequence Properties

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

Related concepts:

- [Step Groups](#)
- [Sequence Local Variables](#)
- [Sequence File Global Variables](#)
- [Sequence Parameters](#)

- [Callback Sequences](#)
- [Sequence Files](#)
- [Executing Sequences in Parallel](#)

Sequence Files

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

TestStand includes the following types of sequence files:

- Normal sequence files—Most sequence files you create are normal sequence files that include sequences to test UUTs.
- Process model sequence files—Normal sequence files specify that they always use the station process model, a specific process model, or no process model.
- Callback sequence files—TestStand uses callback sequences under specific circumstances. Sequence files can contain sequences that override these callbacks. Usually, an application has one Station callback sequence file and one Front-End callback sequence file.

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

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

Viewing the Hierarchy of Sequence Calls

Use the Sequence Hierarchy window in the TestStand Sequence Editor to display a graph of the sequence call and callback sequence relationships among sequence files and sequences to more easily visualize, navigate, and maintain test sequences.

Comparing and Merging Sequence Files

The TestStand File Diff and Merge utility is a stand-alone application you can use to compare and merge non-type differences between two or three sequence files and

compare type differences among two sequence files or type palette files.

Password-Protecting Sequence Files

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

Analyzing Sequence Files

Use the TestStand Sequence Analyzer in the TestStand Sequence Editor or the stand-alone sequence analyzer application to analyze projects outside of the TestStand Sequence Editor.

Related concepts:

- [Sequences](#)
- [Normal Sequences](#)
- [TestStand Process Models](#)
- [Callback Sequences](#)
- [Type Concepts](#)
- [TestStand UI Controls](#)
- [Executing Sequences in Parallel](#)
- [Sequence File Global Variables](#)

Step Groups

TestStand executes the steps in the following step group order:

1. **Setup**—Typically contains steps that initialize instruments, fixtures, or a UUT.
2. **Main**—Typically contains the bulk of the steps in a sequence, including the steps that test the UUT.
3. **Cleanup**—Typically contains steps that power off or restore the initial state of instruments, fixtures, and the UUT.

Use separate step groups to ensure that the steps in the Cleanup step group execute regardless of whether the sequence completes successfully or whether a run-time error occurs in the sequence. If a step in the Setup or Main step group causes a run-time error to occur or if the operator terminates the execution, the flow of execution stops and jumps to the Cleanup step group. Steps in the Cleanup group always run even when some of the steps in the Setup group do not run. When a step in the Cleanup group causes a run-time error, execution continues to the next step in the Cleanup group.

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

Sequence Local Variables

Use local variables for storing data relevant to the execution of the sequence, such as maintaining counts or holding intermediate values. You can pass local variables by value or by reference to any step in the sequence that calls a subsequence or any step that calls a code module that uses a module adapter. You can also access local variables from code modules of steps in the sequence using the TestStand API.



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

View and edit the local variables in the Variables pane of the Sequence File window. Use the Value column in the Variables pane to modify string, numeric, and Boolean values.

You can create an unlimited number of local variables in a sequence.

Related concepts:

- [Variables](#)
- [Code Modules](#)

- [Module Adapters](#)
- [TestStand ActiveX API Overview](#)

Sequence File Global Variables

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

Sequence Parameters

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

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

View and edit the parameters for a sequence on the Variables pane of the Sequence File window. Use the Value column on the Variables pane to modify string, numeric, and Boolean values.

NI recommends that you carefully analyze use cases and create extensible, parameterized code modules when you develop a TestStand system so you can more easily reuse the code modules with different parameters for future systems. However, code modules that are highly parameterized might be difficult to use and maintain.

Related concepts:

- [Module Adapters](#)
- [TestStand ActiveX API Overview](#)

Steps

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

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

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

Related concepts:

- [Code Modules](#)
- [Built-In Step Properties](#)
- [Custom Step Properties](#)
- [Module Adapters](#)
- [Updating the Status Bar Using UI Messages](#)
- [TestStand Building Blocks](#)

Step Types

Just as each property or variable has a data type, each step has a step type. Each step of a type includes the built-in step properties and any number of custom step properties the step defines. Although all steps of the same type have the same properties, the values of those properties can differ. The step type specifies the initial values of all the step properties.

You can create a test application using only the predefined step types, and you can also create custom step types to define standard, reusable classes of steps that apply specifically to the application.

Source Code Templates

You can define a source code template for a new step type. When you create a new step of a particular type, you can use a source code template to generate source code for the code module of the step. You can also create additional code templates for built-in step types when you create a new step type.

Multiple Code Templates per Step Type

You can specify more than one code template for a step type. For example, you might want to have code templates that contain example code for conducting the same type of tests with different types of instruments or data acquisition boards. When a step type has multiple code templates and you click the **Create Code** button on the Module tab of the Step Settings pane, TestStand prompts you to select from a list of templates or uses the template you selected on the Module tab when it exists.

Related concepts:

- [Built-In Step Types](#)
- [Built-In Step Properties](#)
- [Custom Step Properties](#)
- [Creating Custom Step Types](#)
- [Code Modules](#)
- [Instrument Control Step Types](#)

Built-In Step Properties

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

- General Panel
- Run Options Panel
- Looping Panel
- Post Actions Panel
- Switching Panel
- Synchronization Panel
- Expressions Panel
- Preconditions Panel

- Requirements Panel
- Additional Results Panel
- Property Browser Panel

Related concepts:

- [Accessing Properties Using API](#)
- [Custom Step Properties](#)

Custom Step Properties

You can usually modify the values of custom step properties using the tabs on the Step Settings pane. If the step type does not include a tab for the custom properties, select the Property Browser panel to view the custom properties for the step.

Although code modules usually do not modify the values of the built-in step properties at run time, they often modify and read the values of the custom step properties when determining the pass/fail status of the step.



Note The Property Browser is read-only during execution.

Related concepts:

- [Code Modules](#)
- [Built-In Step Properties](#)
- [Accessing Properties Using API](#)

Code Modules

A code module is a program module from an application development environment (ADE) or programming language and can contain one or more functions that perform a specific test or other action. Module adapters load and call code modules, pass parameters to code modules, and return values and status from code modules.

Configure a step to use a module adapter to a step to specify what type of code module the step calls. For example, use the LabVIEW Adapter to call a LabVIEW VI (.vi) or use the LabWindows/CVI Adapter to call a Microsoft Windows dynamic link

library (.dll).

Module adapters load and call code modules, pass parameters to code modules, and return values and status from code modules.



Note

- Use relative paths to specify code modules because you typically organize code module files in a folder relative to the TestStand sequence file and TestStand steps. Absolute paths can cause errors in deployment and multiple-developer scenarios because directory structures frequently vary among systems. Absolute paths also increase the maintenance burden if you want to move files within the TestStand system.
- Not all steps call code modules. Some steps perform standard actions you configure using panes and dialog boxes. In this case, the panes and dialog boxes use the custom step properties to store the configuration settings you specify.

Related concepts:

- [Module Adapters](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Programming with the TestStand API in LabWindows/CVI](#)
- [Deploying TestStand Systems](#)
- [Custom Step Properties](#)
- [Communicating with Hardware Drivers](#)
- [Executing Code Modules in Parallel](#)

Communicating with Hardware Drivers

NI does not recommend calling functions in hardware driver DLLs directly from TestStand. Instead, NI recommends that you create a code module from a supported application development environment (ADE) to interface with the driver and acquire data. You can then call this code module from a step in a TestStand sequence.

NI recommends this approach for the following reasons:

- Writing a code module in a supported ADE provides access to driver wrapper code, such as the LabVIEW NI-DAQmx Library, which is fully documented and supported.
- Using a code module can help modularize components of the test system and reduce unnecessary complexity in a TestStand sequence. For example, instead of creating multiple TestStand steps that interact directly with a hardware driver, you can create a single code module that configures the hardware, acquires measurement data, and returns a numeric result to TestStand for evaluation.

Properties

A property can store a single value or a multidimensional array of values of the same data type.

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

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

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

Related concepts:

- [Built-In and Custom Properties](#)

Built-In and Custom Properties

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

access the built-in properties through the TestStand API.

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

Related concepts:

- [Built-In Step Properties](#)
- [TestStand ActiveX API Overview](#)
- [Custom Step Properties](#)

PropertyObject Attributes

You can create attributes for TestStand PropertyObjects, such as variables, steps, module parameters, and type definitions. Use attributes to store additional information about a PropertyObject within a sequence file or type palette file, such as the author of a sequence. Every PropertyObject includes an Attributes property, which is an unstructured container that stores arbitrary data.

Use container subproperties under the Attributes property to serve as attribute namespaces. NI recommends that you create a uniquely named container or hierarchy of containers under which you store attribute variables to avoid name conflicts, such as `CompanyName.AttributeCategory.Attribute`. The name NI is reserved at the root level for National Instruments.

Editing Attributes

Select **Advanced»Edit Attributes** from the Sequences pane context menu to launch the Attributes dialog box, in which you can create and edit attributes for the sequence. When the sequence includes attributes, you can click the **Edit Attributes** button in the Sequence column of the Sequences pane to launch the Attributes dialog box.

Right-click a PropertyObject on the Variables pane and select **Advanced»Edit Attributes** from the context menu to launch the PropertyObject Attributes dialog box, in which you can create and edit attributes. PropertyObjects with attributes include an Edit Attributes button in the Name column of the Variables pane.

Click the **Edit Attributes** button on the General panel of the Properties tab on the Step

Settings pane to launch the Attributes dialog box for steps. Steps with attributes include the word `Attributes` in the step description.

Right-click a parameter on the Module tab of the Step Settings pane in the sequence editor or in the Specify Module dialog box in user interfaces and select **Edit Attributes** from the context menu to launch the Attributes dialog box for a module parameter. Parameters with attributes include an Edit Attributes button in the Name column of the Parameters Table. When editing a LabVIEW or .NET code module, you can edit attributes for fields of containers and elements of arrays. TestStand associates parameter attributes with the code module parameter, which TestStand stores with the step configuration information, not with the parameter value that TestStand passes to the code module.

Managing Attributes

Right-click a type in the Types window of the sequence editor, select **Properties** from the context menu, click the **Advanced** button on the General tab of the Properties dialog box, and select **Attributes** to launch the Attributes dialog box for a type definition. When editing the attributes of a type definition, you can edit both attributes and type attributes. For a type definition, attributes determine the default values of the attributes of type instances. Type attributes are attributes for the type definition and do not exist for type instances.

Enable the **Include Attributes** option on the Contents tab of the Report Options dialog box or enable the `PropFlags_IncludeInReport` flag on both an attribute and on the `PropertyObject` object that owns the top-level attribute to include attribute information in reports. Typically, a report generator does not recursively apply the `PropFlags_IncludeInReport` flag from a property to its attributes.

The default TestStand database schemas do not log attribute values for result properties. You can customize a database logging schema to log attributes.

Enable the **Attributes** option on the Filter tab of the Find/Replace in Files dialog box and in the Find/Replace dialog box to include the attributes and type attributes of variables and properties in search results.

The TestStand File Diff and Merge utility detects and merges attribute differences. Right-click an item and select **Advanced»Edit Attributes** from the context menu or

click the **Edit Attributes** button in the Name column of the utility to launch the related Attributes dialog box. When you right-click an item and select **Replace Item Values in File <filename>** from the context menu, TestStand also copies attribute values.

You can specify an attribute or type attribute using a lookup string. Use `Attributes` in a lookup string to specify the attributes of the property object and use `TypeAttributes` to specify the type attributes of the property object. For example, the following lookup string accesses an attribute of a local variable from a sequence:

```
"Locals.Foo.Attributes.AttributeNamespace.Attribute1"
```



Note

- If a property object has a subproperty named `Attributes` or `TypeAttributes`, the subproperty takes precedence over the attributes or type attributes in a lookup string. As a result, some TestStand features, such as accessing attributes using lookup strings, undo, and find and replace, do not support attributes if the parent property object also has a subproperty named `Attributes`. NI does not recommend using attributes and a subproperty named `Attributes` on the same property object. You can convert the subproperty to attributes or rename the subproperty.
- You cannot specify an attribute or type attribute with a lookup string you pass to a method in the TestStand API if you also use the `PropOption_InsertIfMissing` option in the method call. Using the `PropOption_InsertIfMissing` option with a lookup string that contains `Attributes` or `TypeAttributes` causes methods in the TestStand API to create a subproperty for `Attributes` or `TypeAttributes`.

Related concepts:

- [Default TestStand Table Schemas](#)
- [Logging Attribute Values](#)

Variables

Variables are properties you can freely create in certain contexts. Variables can apply globally to a sequence file or locally to a particular sequence. You can also use station global variables with values that persist across different executions and across different invocations of the sequence editor or user interfaces. The TestStand Engine maintains the value of station global variables in the `StationGlobals.ini` file in the `<TestStand Application Data>\Cfg` directory.

You can use TestStand variables to share data among tests written in different programming languages, even if the data representations are incompatible. You can pass values you store in variables and properties to code modules. You can also use the TestStand API to access variable and property values directly from code modules. Each step in a sequence can include properties. The type of step determines its set of properties.

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

Related concepts:

- [Station Global Variables](#)
- [TestStand ActiveX API Overview](#)
- [Step Types](#)
- [Sequence Local Variables](#)

Station Global Variables

Station global variables are variables specific to a computer that you can access from all steps and sequences on that computer. Any changes you make to the value of a station global variable are reflected in all other steps and sequences.

Use the Station Globals window of the TestStand Sequence Editor to create station global variables. Creating a station global variable is similar to creating a variable of

any other scope. You can access and use station global variables similarly to variables of other scopes, such as local variables.



Note When you use station global variables in TestStand, you might encounter problems with concurrent access, which is similar to the behavior of using global variables in other programming languages. Use queues and notifications for intrathread communication to work around these issues.

While running, TestStand stores the station global variables in the `StationGlobals` property object.

TestStand saves the station global variables in the `StationGlobals.ini` file in the `<TestStand Application Data>\Cfg` directory automatically when you close the TestStand Engine. You can also save the station global variables from the `Station Globals` window or by calling the `Engine.CommitGlobalsToDisk` method.



Note Until you save the station global variables, new instances of TestStand do not have access to the new station global variables. Any changes you make to the station global variables in one instance of TestStand do not appear in any other instances of TestStand that are running at the same time.

The TestStand Deployment Utility does not automatically deploy the station global variables. You can include the `StationGlobals.ini` file in the deployment image, but doing so might remove any station global variables that exist on the target computer to which you deploy the image.



Note You can programmatically create variables under the property that `Engine.TemporaryGlobals` returns. TestStand does not persist these objects to file. You can also use `Execution.RunTimeVariables` and `Thread.RunTimeVariables` to programmatically create variables at runtime for an execution or for a thread in an execution.

Related concepts:

- [Sequence Local Variables](#)

- [Station Globals TS Subproperty](#)
- [<TestStand Application Data> Directory](#)

Expressions

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

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

The following is an example of an expression:

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

All TestStand controls that accept expressions provide context-sensitive editing features, such as drop-down lists, syntax checking, and expression coloring to help you create expressions. At any point while editing an expression, you can press <Ctrl-Space> to show a drop-down list of valid expression elements.

The expression editor includes visual assistance for finding and manipulating nested delimiters such as parenthesis, brackets, and braces.

- Place your cursor to the left of a delimiter and the expression editor makes the matching pair bold.
- Type <Ctrl +]> to toggle the cursor between the matching delimiters.
- Type <Ctrl + Shift +]> to select the matching delimiters and everything between them.

Related concepts:

- [Expression Operators](#)
- [TestStand ActiveX API Overview](#)
- [Statement Step Type](#)

Expression Operators

Table 6. Expression Operators

Operator Class	Operators in Symbol Form
Arithmetic	The arithmetic symbols include the following items: +, -, *, /, MOD, %, ++, and --.
Assignment	The assignment symbols include the following items: =, +=, -=, *=, /=, %=, ^=, &=, and =. The equal operator evaluates the expression on the right side of the operator and sets the operand on the left side of the operator to the result. When operating on container properties, the hierarchy of subproperty names on the left and right side must match. For each property value, the assignment converts between different data types whenever possible, for example, numeric to string.
Bitwise	The bitwise symbols include the following items: AND, OR, NOT, XOR, &, , ~, ^, >>, and <<.
Comparison	The comparison symbols include the following items: ==, !=, <>, >, >=, <, and <=. Floating-point comparisons use 14 digits of precision. Comparison operators work on string, numeric, and Boolean values. If one operand is a string and another is a numeric, the operator converts the string to a numeric value. If both operands are strings, the comparison is case-insensitive. If both operands are non-zero floating-point numeric values, the comparison uses 14 digits of precision. TestStand treats NAN and IND as equivalent in expression comparisons.
Logical	The logical symbols include the following items: &&, , and !.
Other	Some additional operators include the following items: () Parentheses—Alter evaluation order. . Dot—Property field separator.

Operator Class	Operators in Symbol Form
	.. Double Dot—Specify a range of indexes in array subscripts. This operator indexes multiple elements in the array and returns a subarray. The operand on each side of the range operator must evaluate to a numeric value, such as <code>array[1..3]</code>. [] Brackets—Array subscript  Note 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 that contains the name of a step or sequence element in the array, such as <code>RunState.Sequence.Main["MyGoto"]</code>. You can use the double dot operator (<code>..</code>) within the <code>[]</code> operator to index multiple elements in the array and return a subarray. For example, the expression <code>array[1..3][2]</code> returns the elements at indexes <code>[1][2]</code>, <code>[2][2]</code>, and <code>[3][2]</code> in a new subarray. , Comma—Expression separator or terminator. ?: Conditional—Given a Boolean value, chooses one of two other expressions to evaluate Usage: <code>booleanValue ? expr1 : expr2</code> { } Array constant. // Single line comment (C++). ' Single line comment (Basic). /* */ Comment (C/C++). & Get Reference—Returns a Reference that refers to the

Operator Class	Operators in Symbol Form
	operand. * Dereference—Returns the object to which the Reference operand refers. The Reference must refer to an object that supports the TestStand PropertyObject interface. -> Dereference and access field—Returns the specified subproperty of the object to which the Reference operand refers. The Reference must refer to an object that supports the TestStand PropertyObject interface.
Constants	The formats for the different types of constants include the following items: 1.23e-4 Floating point 1234 Integer 0x1234efa9 Hexadecimal integer 0b11011011 Binary number 1234i64 64-bit integer constant 1234ui64 64-bit unsigned integer constant True Boolean False Boolean "1234wxyz" String @ "file path" Unescaped string, such as @"C:\Windows\temp" Nothing Empty ActiveX reference NAN Not a number IND Indeterminate number

Operator Class	Operators in Symbol Form
	INF Infinite number PI PI constant (3.141592...) tsRed Color constant  Note TestStand declares some special constant values that are similar to values used in other environments or used infrequently.

Related concepts:

- [Levels of Precedence in Operators](#)
- [Special Constant Values](#)

Levels of Precedence in Operators

Expression Type	Operator	Example
primary	Literal Identifier (expression) {element1, element2, ...}	3.14 or Locals.String (Seconds() / 1000) {1.0, 2.5, 5.0}
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

Expression Type	Operator	Example
range	<code>..</code>	<code>Locals.Array[1..3]</code>
relational	<code><, >, <=, >=</code>	<code>Locals.Number <= 0.1</code>
equality	<code>==, <>, !=</code>	<code>Locals.Number == 2.0</code>
bitwise AND	<code>&, AND</code>	<code>Locals.Number & 0xFFFF</code>
bitwise exclusive OR	<code>^, XOR</code>	<code>Locals.Number ^ 0xFFFF</code>
bitwise inclusive OR	<code> , OR</code>	<code>Locals.Number 0x0008</code>
logical AND	<code>&&</code>	<code>Locals.Bool && Step.Result.PassFail</code>
logical OR	<code> </code>	<code>Locals.Bool Step.Result.PassFail</code>
conditional	<code>? :</code>	<code>Step.Result.PassFail ? 5.0 : 6.0</code>
assignment	<code>=, +=, -=, *=, /=, %=, &=, ^=, =, <<=, >>=</code>	<code>Locals.Number += 2.0</code>
comma	<code>,</code>	<code>Locals.Number1 = 5.0, Locals.Number2 = 6.0</code>

None

Special Constant Values

TestStand declares a number of constant values. Some values, like the keywords `True` and `False`, are similar to values used in other environments. Other values are used less often. The following table lists all known constant definitions and their common uses.



Note TestStand is not case sensitive for constant names. For example, `true` is the same as `True` and `TRUE`.

Table 7. Special Constant Values

Constant	Definition	Use
1.23e ⁻⁴	Floating Point Constant	Range approximately 1.7E⁻³⁰⁸ to 1.7E⁺³⁰⁸ with 15 significant digits.  Note The Use Localized Decimal Point option on the Localization tab of the Station Options dialog box and your operating system settings determine the decimal point character you use in a floating point number.
1234, 0xFFFF, 0b1011	32-bit Integer Constant	Range -2147483648 to 4294967295. TestStand supports hexadecimal and binary formats with this type of constant with the use of the 0x and 0b prefixes. TestStand treats hexadecimal and binary versions of these constants as unsigned values with a range of 0 to 4294967295. TestStand converts and represents these constants internally as double-precision, floating-point numbers.
1234i64, 0xFFFFi64, 0b1011i64	64-bit Signed Integer Constant	Range -9223372036854775808i64 to 9223372036854775807i64. TestStand supports hexadecimal and binary formats with this type of constant with the use of the 0x and 0b prefixes. However, if the high order bit is set, TestStand interprets the constant as a negative value.
1234ui64, 0xFFFFui64, 0b1011ui64	64-bit Unsigned Integer Constant	Range 0ui64 to 18446744073709551615ui64. TestStand supports hexadecimal and binary formats with this type of constant with the use of the 0x and 0b prefixes.
True/False	Boolean Constant	True or False.
"1234wxyz"	String Constant	Refer to Special String Characters for more information.
@ "C:\Windows\temp"	Unescaped String Constant	Add an @ before the enclosed quotes to turn a string constant into an unescaped string constant. Unlike for regular string constants, backslashes are left as is instead of specifying an escape sequence. Use "" for " within an unescaped string constant.
Nothing	Object Reference Constant	You can determine whether an object reference refers to an object by comparing the reference to Nothing. You can release the object to which an object reference refers by setting the value of the

Constant	Definition	Use
		reference to Nothing in an expression. TestStand handles the garbage collection of references set equal to Nothing when it retrieves all memory that has gone out of scope. This reference is the equivalent of Nothing in Microsoft Visual Basic .NET, null in C#.NET, or NULL in LabWindows/CVI and Microsoft Visual C++.
PI	PI Constant (3.141592...)	The ratio of the circumference of a circle to its diameter.
NAN	Not a Number	A numeric value that is not a valid number. The value is equivalent to a quiet NaN value. TestStand also displays NAN for signaling NaN values. IND is a special kind of quiet NaN value.  Note TestStand treats NAN and IND as equivalent in expression comparisons.
IND	Indeterminate Number	A numeric value that is indeterminate. The value is equivalent to a special kind of quiet NaN value. Some numeric operations, such as Sqrt(-1), will result in an indeterminate value.  Note TestStand treats NAN and IND as equivalent in expression comparisons.
INF	Infinite Number	A numeric value that is infinite or negative infinite. Equivalent to an INF value in Visual C++.



Note The Use Localized Decimal Point option on the Localization tab of the Station Options dialog box and your operating system settings determine the decimal point character you use in a floating point number.

Special String Characters

When located within the quotes of a string, the following characters are considered known escape sequences. The following table describes how TestStand interprets these characters.

Table 8. Special String Characters

String Character	Interpretation in String
\\	Backslash
\n	Line feed
\r	Carriage return
\xNN	Hexadecimal character code
\NN	LabVIEW-style hexadecimal character codeValid only for uncompressed binary data.
\"	Quote
\t	Tab



Note

- Always enclose strings in quotes.
- String literals in TestStand expressions use a backslash as the escape character, similar to other programming languages like C/C++ and C#. Therefore, if you specify a path using a string literal in an expression, such as for the parameter value of a code module, you must use two backslash characters for each actual backslash the string value requires because TestStand interprets the first backslash character as escaping the character after it. If the character after the first backslash has a special meaning, such as \n (new line) or \t (tab), TestStand replaces the backslash character and the character which follows it with the corresponding special character code. However, if you are storing a path directly in a string variable programmatically or by using the Variables view, use one backslash character per actual backslash because, unlike string literals in expressions, the value of a string variable is not considered escaped and TestStand does not interpret the backslash characters in such strings as having any special meaning.

Process Models

Testing a UUT requires more than just executing a set of tests. Usually, the test system must perform a series of operations before and after it executes the sequence that performs the tests. Common operations of the process model include identifying the UUT, notifying the operator of pass/fail status, logging results, and generating reports. The set of operations and the flow of execution is called a process model. A TestStand process model is a sequence file you can use to define standard testing operations so you do not have to re-implement the same operations in every sequence file you write.

TestStand includes predefined Sequential, Parallel, and Batch models you can modify or replace. Use the Sequential model to run a test sequence on one UUT at a time. Use the Parallel and Batch models to run the same test sequence on multiple UUTs simultaneously.

You can modify an existing TestStand process model or you can create a custom process model. The ability to modify a process model is essential because the testing process can vary depending on production lines, production sites, or company systems and practices. You can edit a process model in the same way you edit other sequence files. You can also use client sequence files to customize various model operations by overriding the callback sequences process models define.

Related concepts:

- [Configuring Logging to Custom Database](#)
- [TestStand Process Models](#)
- [Sequential Process Model](#)
- [Parallel Process Model](#)
- [Batch Process Model](#)
- [Modifying Process Model Sequence Files](#)
- [Process Model Architecture](#)
- [Process Model Plug-In Architecture](#)
- [Process Model Support Files](#)

Specifying Process Model Files

The station process model file is the process model file TestStand uses for all sequence files on the current test station.

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

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

Main Sequence and Client Sequence File

The Main sequence initiates the tests on a UUT. The process model defines what is constant about the testing process, and Main sequences define the unique steps for the different types of tests to run. When you create a new sequence file, TestStand automatically inserts a Main sequence in the file. You must name each Main sequence `MainSequence`.

The process model invokes the Main sequence as part of the overall testing process. TestStand uses the station process model file to determine which process model file to use with the Main sequence unless the sequence file specifies a different process model file and you enabled the Allow Other Models option in the Station Options dialog box to allow sequence files to override the station model setting.

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

Entry Points

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

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

Related concepts:

- [Entry Point Sequences](#)
- [Using Execution Entry Points](#)
- [Configuration Entry Points](#)

Standard and Custom Data Types

When you create a variable or property, you specify its data type. In some cases, you use a built-in data type, such as a number or a Boolean. In other cases, you might want to use an arbitrarily complex data structure by defining a custom named data type you can reuse with other variables or properties.

When you define a custom named data type, the data type must use a unique name. You can add or delete subproperties in each named data type you create without restriction. For example, you might create a Transmitter data type that contains subproperties such as `NumChannels` and `PowerLevel`.

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



Note Modifying the standard named data types might result in type conflicts when you open other sequence files that reference these types.

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

Related concepts:

- [Data Types](#)
- [Modifying Types](#)

Automatic Result Collection

TestStand can automatically collect the results of each step. You can enable or disable result collection for a step, a sequence, an execution, or for the entire test station. You can generate a report and log results to a database to process the collected test

results.

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

Related concepts:

- [Result Collection](#)
- [Database Logging](#)

Callback Sequences

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

TestStand defines Model callbacks, Engine callbacks, and Front-End callbacks based on the entity that invokes the callback and the location in which you define the callback, as shown in the following table.

Callback Type	Where You Define the Callback	What Calls the Callback	Description
Model Callbacks	The process model file defines Model callbacks, and the client sequence file or <code>StationCallbacks.seq</code> can override the callback	Sequences in the process model file	Customizes the behavior of a process model for each Main sequence that uses it.
Engine Callbacks	<code>StationCallbacks.seq</code> for Station Engine callbacks, the process model file for Process Model Engine callbacks, or a regular sequence file for Sequence File Engine callbacks.	The TestStand Engine defines and invokes Engine callbacks at specific points during execution.	Engine
Front-End	<code>FrontEndCallbacks.seq</code>	User interface application	User interface programs call Front-End callbacks

Callback Type	Where You Define the Callback	What Calls the Callback	Description
Callbacks			so multiple user interfaces share the same behavior for a specific operation.

Related concepts:

- [List of Engine Callbacks](#)
- [Customizing Process Models and Callbacks](#)

Workspaces

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

Use project files to organize related files and directories of files in the test system. You can insert any number of files into a project. For example, you can use a project file to organize a sequence file, code module source files, and other supporting files.

When you use a workspace, you must define at least one project. However, defining multiple projects in a workspace can help you better organize a test system. For example, you can define product model projects that contain all the tests required for the specific product model. When a model becomes obsolete, you can remove only the corresponding project from the workspace to update the test system.

Use workspace files to help organize and access development files, to perform source code control (SCC) operations in the TestStand Sequence Editor or in a TestStand User Interface, and to store debugging information, such as breakpoints and watch expressions.

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

Source Code Control

Use workspace files early in development to easily access files in a source code control (SCC) system. When TestStand loads a workspace, the properties of a workspace specify whether TestStand attempts to connect to an SCC provider. When TestStand successfully connects to an SCC provider, you can use the commands in the Workspace context menu and the Source Control menu to check out and check in files from SCC. If an SCC provider returns warning or error messages when performing SCC operations, the sequence editor displays the messages in the Output pane.

To perform SCC operations on files from within TestStand, select an SCC provider on the Source Control tab of the Station Options dialog box and configure the SCC settings for the workspace on the Source Control tab of the Workspace Object Properties dialog box.



Note TestStand integrates with any 32-bit SCC provider that supports the Microsoft Source Code Control Interface. 64-bit TestStand supports 32-bit SCC providers only and does not support 64-bit SCC providers.



Note Use the Git pane in the TestStand Sequence Editor to connect to Git repositories and perform common version control operations.

Inserting Code Modules in a Workspace

You can right-click a sequence file in the workspace and select **Insert Code Modules** from the context menu to automatically include the code modules a sequence file uses. However, if you are using a workspace only to create a deployment, NI does not recommend this technique.



Note TestStand does not dynamically update code modules. If you add or remove any code modules from your sequence files, you must delete the `CodeModules` folder and then select **Insert Code Modules** again from the Workspace Browser context menu to add the code module files back to the workspace.

Refreshing the Code Modules Directory in a Workspace

When a sequence file in a workspace changes or if you are using a workspace to interface with an SCC system, complete the following steps to manually refresh the code modules directory to avoid deploying files that the test system no longer requires.

1. Delete the current code modules directory from the workspace.
2. Right-click the sequence files for which you want to add code modules.
3. Select **Insert Code Modules** from the context menu.

System Deployment

You can use a workspace and project files to specify all the required files the TestStand Deployment Utility collects to successfully distribute a test system to a target computer. The deployment utility creates an image or an installer for the test system.

Related concepts:

- [Selecting a Source Code Control Provider](#)
- [Using a TestStand Workspace File to Create a Deployment](#)
- [Deploying TestStand Systems](#)
- [Creating a Deployable Image](#)
- [Building a Customized MSI-based Installer](#)

Related information:

- [Git Pane](#)

Selecting a Source Code Control Provider

Microsoft Source Code Control (MSSCC)-Based Providers

TestStand supports several third-party source code control (SCC) providers. Available SCC operations in TestStand are the same regardless of the third-party provider you select. Specific support or functionality for each operation varies by provider.

In some cases, you might decide to use a specific SCC provider because your company

has standardized on that application. If not, you must decide which provider you want to use for managing files. Consult the SCC administrator at your company to find out if you must use a specific provider.

After you select and install an SCC provider, you must configure TestStand to work with that provider. You can configure a TestStand workspace to work with only one SCC provider at a time.

TestStand integrates with any 32-bit SCC provider that supports the Microsoft Source Code Control Interface. 64-bit TestStand supports 32-bit SCC providers only and does not support 64-bit SCC providers.

NI supports the following third-party SCC providers with TestStand:

- Helix Core
- SVN (PushOK plugins)

The following third-party SCC providers are allowed but not supported:

- IBM Rational ClearCase
- MKS Source Integrity
- Serena Version Manager (PVCS)
- Microsoft Visual SourceSafe

Git-Based Providers

The TestStand Sequence Editor supports source code control using Git-based providers through the Git pane. You can connect to your Git repositories and perform common version control operations using the Git pane, and can also work with local Git repositories without using a Git provider. You can use the Git pane to work with any source folder in a Git repository without the need to create a TestStand workspace file.

NI supports the following Git providers in TestStand:

- Microsoft Azure DevOps
- GitHub
- BitBucket

Related concepts:

- [Workspaces](#)

Related information:

- [Git Pane](#)

TestStand File Extensions

The following table includes common file types that TestStand creates and uses to save data.

Table 9. TestStand File Extension Descriptions

File Extension	Description
.c	C source file.
.dll	Dynamic Link Library file.
.html	HyperText Markup Language file. TestStand can use the HTML file format for test reports.
.ini	Standard Microsoft Windows configuration settings file. Used to store most TestStand configuration information.
.llb	LabVIEW library file.
.lvclass	LabVIEW class library file. The class library file defines a new data type.
.lvlibp	LabVIEW packed project library. LabVIEW packed libraries package multiple files into a single file.
.lvproj	LabVIEW project.
.seq	Sequence file. Sequence files are the files in which TestStand applications are written.
.tsaproj	TestStand Sequence Analyzer project. Specifies the files to analyze and the rules and related settings to use during the analysis.
.tpj	TestStand Project file. A project file can store and organize the set of files used in a TestStand application.
.tsd	TestStand Deployment Utility deployment file. Contains the information to build a full deployment or a patch deployment.

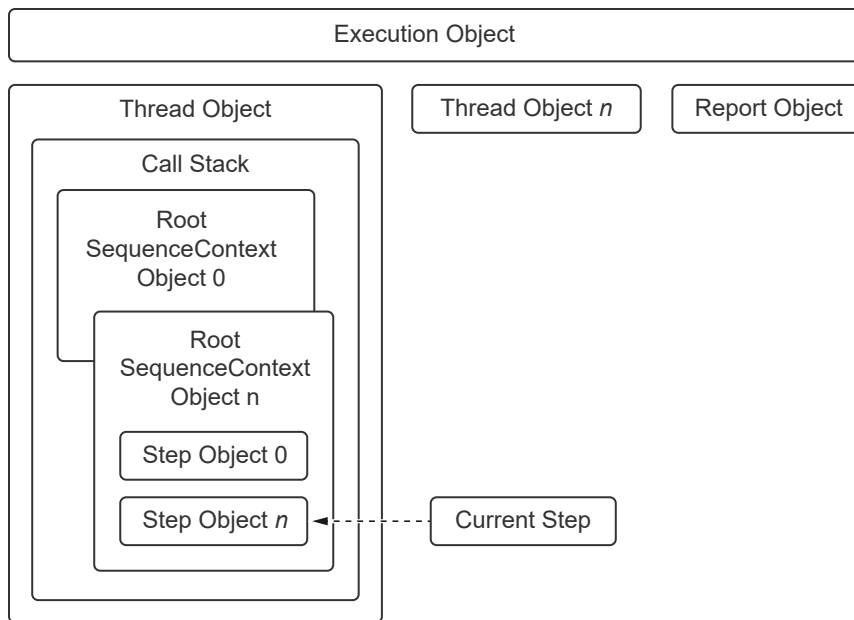
File Extension	Description
.tso	Workspace options file. TestStand saves information in this file to preserve certain user settings, such as window positions, source code control information, and the expansion state of the Workspace pane.
.tsr	TestStand raw results file. Stores results in a compressed format for later processing.
.tsw	TestStand workspace file. A workspace file can store TestStand project files.
.txt	Text file.
.vi	LabVIEW virtual instrument.
.xml	Extensible Markup Language file. TestStand can use the XML file format for test reports.
.xsl	Extensible Stylesheet Language file. TestStand uses these files to apply formatting to XML test reports.

Related concepts:

- [Organizing Test Program Files with LabVIEW Projects](#)
- [Deploying TestStand Systems](#)
- [Patching Deployments](#)
- [Generating and Customizing TestStand Reports](#)

Executions

When you run a sequence, TestStand creates an Execution object that contains all the information TestStand needs to run the sequence and subsequences, as shown in the following figure.



When an execution is active, you can start other executions by running the same sequence again or by running different sequences. TestStand does not limit the number of executions you can run concurrently.

An execution can start with a single thread. You can use Sequence Call multithreading options to create additional threads within an execution or to launch new executions. An execution groups related threads so that setting a breakpoint suspends all threads in the execution. When you suspend, terminate, or abort an execution, you stop all threads in the execution.

Multiple instances of a sequence can run at the same time, such as when you call a sequence recursively or when a sequence runs in multiple concurrent threads. For each instance of the sequence, TestStand creates a copy of the sequence parameters, local variables, and custom properties of each step. When a sequence completes, TestStand discards the values of the parameters, local variables, and discards the values of custom properties when you set the value of the Optimize Non-Reentrant Calls to This Sequence option on the General tab of the Sequence Properties dialog box to `False`.

When TestStand begins executing a sequence, it creates a run-time copy of the sequence local variables and the custom properties of the steps in a sequence. If the sequence calls itself recursively, TestStand creates a separate run-time copy of the local variables and custom step properties for each running instance of the sequence. Modifications to the values of local variables and custom step properties apply only to

the run-time copy and do not affect the sequence file in memory or on disk.



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

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

The Execution tab of the Station Options dialog box tab provides a number of execution options that control tracing, breakpoints, and result collection.

Related concepts:

- [Built-In Step Properties](#)
- [Sequences](#)

Sequence Context

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

A sequence context is an object in the TestStand API that represents the execution state of a sequence. For each active sequence, TestStand maintains a sequence context that contains a reference to the run-time copy of the sequence so you can access all the objects, variables, and properties that relate to the execution of the sequence.

The contents of the sequence context vary depending on the currently executing sequence and step, the location of the active sequence in the call stack, and the identity of the execution in which the active sequence resides. Depending on the current state of execution, sequence context subproperties might not exist. When a property exists, the contents of the property can vary.

The sequence context `Step` subproperty is an example of a property that depends on the execution state. During execution, the `Step` property exists only when a step is executing. When execution suspends between steps, the `Step` property does not exist. When it exists, the contents of the `Step` property are the contents of the currently executing step.

Referring to Subproperties

To refer to a subproperty, use a period to separate the name of the property from the name of the subproperty. For example, refer to the `CurrentUser` subproperty in the `TS` subproperty of the `StationGlobals` property as `StationGlobals.TS.CurrentUser`.

Related concepts:

- [Accessing the Sequence Context](#)
- [Top-Level Properties](#)
- [InitialSelection Subproperties](#)
- [Sequence Object Subproperties](#)
- [RunState.Step and Other Step Objects](#)
- [RunState Subproperties](#)
- [SequenceFile Object Subproperties](#)
- [Station Globals TS Subproperty](#)

Accessing the Sequence Context

In expressions, you can access the value of a variable or property by specifying a path from the sequence context to the particular variable or property. For example, you can specify the status of a step using the following expression: `Step.Result.Status = "Passed"`

During an execution, you can view and modify the values of the properties in the sequence context from the Variables pane in the Execution window. The Variables pane displays the sequence context for the sequence invocation currently selected on the Call Stack pane. You can also monitor individual variables or properties from the Watch View pane.

You can pass a reference to a `SequenceContext` object to a code module. In code

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

Select **View»Sequence File»Variables** or **View»Execution»Variables** in the TestStand Sequence Editor to display the Variables pane, which contains the names of variables, properties, and sequence parameters you can access from expressions and code modules.



Note Some properties are not populated until run time.

Related concepts:

- [Sequence Context](#)

Top-Level Properties

Sequence Context Subproperty	Description
<code>Locals</code>	Contains local variables in the current sequence. Only the current sequence can access these variables.
<code>Parameters</code>	Contains parameter variables in the current sequence. Only the current sequence or calling sequences can access these variables.
<code>FileGlobals</code>	Contains file global variables in the current sequence file. All the sequences in the current sequence file can access these variables.
<code>StationGlobals</code>	Contains station global variables on the computer. Any sequence on the current computer can access these variables. Station global variables are stored on disk and the values persist even after you close TestStand. Refer to <code>StationGlobals TS Subproperty</code> for more information about the <code>StationGlobals</code> property.
<code>ThisContext</code>	Contains a reference to the current sequence context. You typically use this property to pass the entire sequence context as an argument to a subsequence or a step code module.
<code>RunState</code>	Contains properties that describe the current state of execution. Refer to <code>RunState Subproperties</code> for more information about the <code>RunState</code> property.
<code>Step</code>	Contains the properties in the currently executing step. The <code>Step</code> property

Sequence Context Subproperty	Description
	exists only while a step executes. The property does not exist when the execution is between steps, such as at a breakpoint.

The following properties in the sequence context refer to objects that exist before, and persist after, the current execution:

- `StationGlobals`
- `RunState.InitialSelection`
- `RunState.SequenceFile`
- `RunState.ProcessModelClient`

Any modifications you make to these objects affect all executions in the current TestStand session. When you save the modifications to disk, they affect future TestStand sessions.



Note Built-in properties of steps are flagged to be shared at run time. Any changes to built-in properties within the run-time copy of the step edits the original Step object in the sequence file.

Related concepts:

- [Station Globals TS Subproperty](#)
- [RunState Subproperties](#)

Station Globals TS Subproperty

The `StationGlobals` property object contains the station global variables for the TestStand 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.

TestStand creates a `TS` subproperty in the `StationGlobals` property to hold the standard station global variables TestStand defines.

Table 10. Sequence Context Subproperties

Sequence Context Subproperty	Description
TS	Contains the TestStand-specific station global variables.
LastUserName	Login name of the user that logged in most recently.
CurrentUser	User object for the currently logged-in user. This property does not exist when no user has logged in. Refer to <i>User and Group Privileges</i> for more information about the User standard data type.

Related concepts:

- [Station Global Variables](#)

RunState Subproperties

The RunState property object contains properties that describe the state of execution in the sequence invocation.

Sequence Context Subproperty	Description
Engine	Engine object in which the sequence invocation executes.
Root	Sequence context for the root sequence invocation. When you initiate an execution using a process model entry point, the property is the sequence context for the entry point. For example, when 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. When 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. When you initiate an execution using the default process model entry point, the Main property is the sequence context of the Main sequence. When you initiate an execution on a sequence without using a process model entry point, the Main property object is the sequence

Sequence Context Subproperty	Description
	context for whichever sequence you run.
<code>ThisContext</code>	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 code module.
<code>Caller</code>	Sequence context for the sequence that called the current sequence. This property does not exist in the root sequence context.
<code>CallingStep</code>	Run-time copy of the Step object for the step that called the current sequence. This property does not exist in the root sequence context. Any changes to property values in this object only modify the execution version of the object.
<code>InitialSelection</code>	Contains references to properties, steps, sequences, sequence files, and executions that are selected or active when you start an execution. When the Sequence File window is active, the InitialSelection property contains non-execution versions of the selected objects. When an Execution window is active, the InitialSelection property contains execution versions of the selected steps, properties, and sequences as well as non-execution versions of the selected sequence file. You usually use this property in custom Tools menu commands to operate on the selected objects in a sequence file.  Note Whenever you make changes to subproperty values in non-execution versions of a <code>PropertyObject</code>, <code>Step</code>, <code>Sequence</code>, or <code>SequenceFile</code> object the InitialSelection property contains, you modify 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 the objects it contains from a code module, you must increment the <code>SelectedFile.ChangeCount</code> subproperty of InitialSelection.
<code>Report</code>	Report object for the execution.
<code>Execution</code>	Execution object in which the sequence invocation runs.
<code>Thread</code>	Thread object in which the sequence invocation executes.
<code>TestSockets</code>	Container that holds the number of test sockets the user has configured to execute, as well as a test socket index unique to each test socket.

Sequence Context Subproperty	Description
SequenceFile	The SequenceFile object for the sequence invocation. Refer to SequenceFile Object Subproperties 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.
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 only modify the execution version of the object. Refer to Sequence Object Subproperties for the contents of the Sequence property.  Note Built-in properties of sequences are flagged to be shared at run time. Changes to built-in properties within the run-time copy of the sequence also edit the original Sequence object in the sequence file.
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. Changes to property values in this object only modify the execution version of the object.  Note Built-in properties of steps are flagged to be shared at run time. Changes to built-in properties within the run-time copy of the step also edit the original Step object in the sequence file.
Step	Run-time copy of the Step object for the currently executing step. This property does not exist when the execution pauses between steps, for example, at a breakpoint. Changes to property values in this object only modify the execution version of the object.  Note Built-in properties of steps are flagged to be shared at run time. Changes to built-in properties within the run-time

Sequence Context Subproperty	Description
	copy of the step also edit the original <code>Step</code> object in the sequence file.
<code>NextStep</code>	Run-time copy of the <code>Step</code> 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. Changes to property values in this object only modify the execution version of the object.  Note Built-in properties of steps are flagged to be shared at run time. Changes to built-in properties within the run-time copy of the step also edit the original <code>Step</code> object in the sequence file.
<code>SequenceError</code>	Container that holds the error code, message, and occurred flag to report to the step that calls the sequence.
<code>ErrorReported</code>	Indicates whether TestStand sent a <code>BREAK ON RTE</code> message to the user interface for the error the <code>SequenceError</code> property stores. Typically, a user interface launches a Run-Time Error dialog box when it receives a <code>BREAK ON RTE</code> message. When a sequence with an error returns, TestStand transfers the value of <code>ErrorReported</code> to the context of the calling sequence. This prevents the calling sequence from generating a duplicate <code>BREAK ON RTE</code> message for the error the subsequence returns.
<code>IsProcessModel</code>	Indicates whether the sequence invocation is a sequence in the process model.
<code>Tracing</code>	Indicates whether tracing is active for the sequence invocation.
<code>SequenceFailed</code>	Indicates whether the current status of the sequence invocation is <code>Failed</code>.  Note Setting this property to <code>True</code> causes the sequence to fail, but the execution does not go to the Cleanup step group when the sequence or the station options specify to immediately go to cleanup on sequence failure. Set the value of <code>RunState.GotoCleanup</code> to <code>True</code> to force a sequence

Sequence Context Subproperty	Description
	 to go to the Cleanup step group.
GotoCleanup	Indicates whether the execution proceeds to the Cleanup step group after completing the currently executing step.  Note Changes to property values in this object only modify the execution version of the object.
StepGroup	Contains the name of the step group—Main, Setup, or Cleanup—the sequence invocation is executing. By modifying the value of this property, you can specify the step group TestStand executes next.  Note Changes to property values in this object only modify the execution version of the object.
CallStackDepth	Zero-based index of the currently executing sequence in the call stack. For example, when 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.
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.  Note This property exists only for steps other than the first step of each step group.
StepIndex	Zero-based index of the currently executing step in the step group. TestStand sets the property 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 property 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 TestStand executes next.

Sequence Context Subproperty	Description
	 Note Changes you make to this property do not immediately affect the value of the <code>RunState.NextStep</code> property object.
<code>LoopIndex</code>	Loop index for the active step in the sequence invocation. By default, steps 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.
<code>NumStepsExecuted</code>	Number of steps that executed in the sequence invocation.
<code>LoopNumPassed</code>	Number of iterations a looping step completes with a status of <code>Passed</code> or <code>Done</code> .
<code>LoopNumFailed</code>	Number of iterations a looping step completes with a status of <code>Failed</code> .
<code>LoopNumIterations</code>	Number of iterations a looping step completes.
<code>ProcessModelClient</code>	The <code>SequenceFile</code> object for the client sequence file of the process model. This property exists only for executions you initiate through a process model entry point.  Note When you save the sequence file, TestStand saves any changes you make to property values in the <code>SequenceFile</code> object.
<code>IsEditor</code>	Indicates whether the current graphical user interface (GUI) is a sequence editor.

Related concepts:

- [InitialSelection Subproperties](#)
- [Sequence Object Subproperties](#)
- [SequenceFile Object Subproperties](#)

SequenceFile Object Subproperties

The following sequence context subproperties are `SequenceFile` objects:

- `RunState.SequenceFile`
- `RunState.ProcessModelClient`
- `RunState.InitialSelection.SelectedFile`

When you save the sequence file, TestStand saves any changes you make to property values in a `SequenceFile` object.

Sequence Context Subproperty	Description
<code>ChangeCount</code>	The number of changes you have made to the <code>SequenceFile</code> object. You must increment this property whenever you modify other <code>SequenceFile</code> object properties from a code module. The increment indicates to the sequence editor that you have changed the sequence file.
<code>LastSavedChangeCount</code>	The value of the <code>ChangeCount</code> property when the sequence editor last saved the sequence file.
<code>Data</code>	Contains the sequences and file global variables in the sequence file.
<code>Seq</code>	Subproperty of <code>Data</code> . Contains an array of all <code>Sequence</code> objects in the sequence file.
<code>File Global Defaults</code>	Subproperty of <code>Data</code> . Contains the default values for the global variables in the sequence file.
<code>Requirements</code>	Contains a <code>Links</code> string array subproperty that represents the product and unit requirements the file covers.
<code>Path</code>	The absolute pathname of the sequence file.

Sequence Object Subproperties

Each `SequenceFile` object in the sequence context contains an array of `Sequence` objects in the `Data.Seq` subproperty. The `RunState.Sequence` subproperty of the sequence context is the `Sequence` object for the current sequence invocation. The `RunState.Sequence` subproperty is a run-time copy of the `RunState.SequenceFile.Data.Seq` array element for the currently executing sequence.

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.SequenceSequence` object. The `RunState.Step` subproperty represents the currently executing step.

Sequence Context Subproperty	Description
 Parameters	Contains the parameters of the sequence. In the <code>RunState.Sequence</code> object, the parameters contain the values the calling sequence passes. In non-execution instances of <code>Sequence</code> objects, the parameters contain their default values.
 Locals	Contains the local variables of the sequence. In the <code>RunState.Sequence</code> object, the local variables contain the current values in the sequence invocation. In non-execution instances of <code>Sequence</code> objects, the local variables contain their default values.
 ResultList	In the <code>RunState.Sequence</code> object, the <code>ResultList</code> local variable contains an array of step results for the sequence invocation. <code>ResultList</code> local variables are empty in non-execution instances of <code>Sequence</code> objects.
 Main	Contains an array of all <code>Step</code> objects in the Main step group.
 Setup	Contains an array of all <code>Step</code> objects in the Setup step group.
 Cleanup	Contains an array of all <code>Step</code> objects in the Cleanup step group.

RunState.Step and Other Step Objects

All `Step` objects include custom properties including a `Result` subproperty, which contains the `Error`, `Status`, and `Common` subproperties.

The sequence context references steps in various locations. The `RunState.NextStep`, `RunState.PreviousStep`, and `RunState.Step` properties contain run-time copies of the `Step` object for the steps that are executing next, previously executed, and currently executing, respectively.

The `Step` objects within the `RunState.Sequence` property contain a run-time copy of the current sequence invocation. The `Step` objects within sequences under the `RunState.SequenceFile.Data.Seq` property contain the edit-time copy of the currently executing sequence file.

All `Step` objects include custom properties and built-in properties. The `Result` built-in

step property contains the `Error`, `Status`, `ReportText`, and `Common` subproperties.

InitialSelection Subproperties

The `RunState.InitialSelection` subproperty specifies the set of properties, steps, and sequences, as well as the file or execution that is selected or active when you start a new execution. The TestStand Sequence Editor automatically specifies the appropriate subproperties of `InitialSelection` when the Station Global, Sequence File, and Execution windows are active. You usually use this property in sequences that custom Tools menu commands or Process Model entry points call.

Sequence Context Subproperty	Description
<code>SelectedSteps</code>	Contains an array of <code>Step</code> objects selected when the execution started. The array is empty for non-root sequence contexts. When the Steps pane of an Execution window is initially active, the array contains execution versions of the <code>Step</code> objects.
<code>SelectedPropertyObjects</code>	Contains an array of <code>PropertyObject</code> objects selected when the execution started. When a sequence file is initially active, the array contains subproperties of a <code>Step</code> object, local variable, file global variable, or station global variable. When the Variables pane of an Execution window is initially active, the array contains the currently selected properties. The array is empty for non-root sequence contexts.
<code>SelectedSequences</code>	Contains an array of <code>Sequence</code> objects selected when the execution started. The array is empty for non-root sequence contexts. When the Steps tab of an Execution window is initially active, the array contains execution versions of the <code>Sequence</code> objects.
<code>SelectedStepGroupByIndex</code>	Contains the index of the step group selected when the execution started. The index values are as follows: 0—Setup step group 1—Main step group 2—Cleanup step group This property exists only in the root sequence context.
<code>SelectedFile</code>	Specifies the <code>SequenceFile</code> object for the active sequence file when the execution started. This property exists only in the root sequence context when a sequence file is initially active, or when the Steps pane of an Execution window is

Sequence Context Subproperty	Description
	initially active.
<code>SelectedPropertyObjectFile</code>	Specifies the <code>PropertyObjectFile</code> object for the active file when the execution started. When a sequence file is initially active, this property is identical to <code>SelectedFile</code> . This property exists only in the root sequence context when a Sequence File window is initially active, or when the Steps pane of an Execution window is initially active.
<code>SelectedExecution</code>	When you select an existing Execution window and start a new execution, this property specifies the Execution object for the existing execution the window shows. This property exists only in the root sequence context when an Execution window is initially active.

Executing Sequences

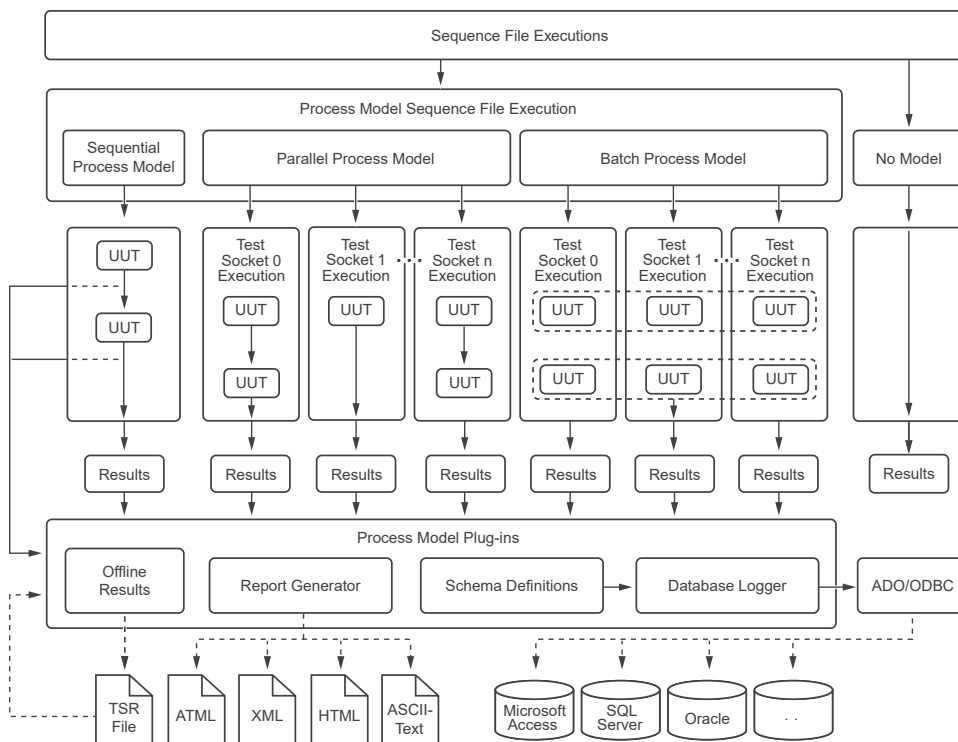
You can initiate an execution in the following ways:

- Launch a sequence through an Execution entry point
- Execute a sequence directly
- Interactively execute a group of steps

During an execution, you can complete the following actions:

- Debug executions
- Terminate or abort executions
- Check for suspended or stopped executions within code modules

The following figure illustrates how sequence files execute.



Related concepts:

- [Using Execution Entry Points](#)
- [Executing a Sequence Directly](#)
- [Interactively Executing Steps](#)
- [Debugging Executions](#)
- [Terminating and Aborting Executions](#)
- [Checking for Suspended or Stopped Execution within Code Modules](#)
- [Executing Sequences in Parallel](#)

Using Execution Entry Points

You can use an Execution entry point to start an execution only when the MainSequence is the sequence open on the Steps pane. The Execute menu of the sequence editor includes a list of Execution entry points.

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

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

Related concepts:

- [Main Sequence and Client Sequence File](#)
- [Entry Point Sequences](#)
- [TestStand Process Models](#)

Executing a Sequence Directly

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

Interactively Executing Steps

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

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

- Run steps interactively from a Sequence File window to create a new execution called a root interactive execution. You can specify station options to control whether the Setup and Cleanup step groups of the sequence run as part of a root interactive execution.
- Run steps interactively from an existing Execution window for a normal execution suspended at a breakpoint. The selected steps run in a nested interactive execution within the context of the normal execution. The steps you run interactively can access the variable values of the normal execution and add to the results. When you use the process model for the original execution, the report

includes these results. When the selected steps complete, the execution returns to the originally suspended step. A configurable station option specifies whether step failures and errors propagate to the original execution.

In interactive mode, the selected steps run in the order in which they appear in the sequence. To configure TestStand to evaluate preconditions when executing interactively, select **Configure»Station Options** and enable the **Evaluate Preconditions** option in the Interactive Executions section on the Execution tab of the Station Options dialog box. You can also use the Branching Mode control in this dialog box to configure whether interactive executions branch to unselected steps.

Related concepts:

- [Step Groups](#)

Debugging Executions

Use the commands in the Debug menu to suspend executions, control how TestStand single-steps within executions, set breakpoints, and stop executions prematurely. Because the **Allow Break While in Code Modules** option on the Execution tab of the Station Options dialog box is enabled by default, executions suspend even if some threads are executing code inside a code module. Using the `Thread.ExternallySuspended` property results in the same behavior.

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

- **Threads and Call Stack panes**—The Threads pane displays the threads running in the execution and selects the active thread to view. The Call Stack pane displays the sequence invocations for the active thread and selects the active sequence invocation to view. TestStand updates these panes while tracing.
- **Variables pane**—Displays the sequence context for the sequence invocation currently selected on the Call Stack pane when executions suspend. Use the Variables pane to examine and modify the values of the variables and properties the sequence context contains.
- **Watch View pane**—Displays the values of the watch expressions you enter. The TestStand Sequence Editor updates the values on the Watch View pane when execution suspends at a breakpoint. When you enable tracing, the sequence editor

also updates the values after executing each step and highlights values that change in red.



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

- **Output pane**—Displays generic messages, warnings, and error messages. By default, the Output pane is empty. Use the `OutputMessage` expression function or the `Engine.NewOutputMessage` method and the `OutputMessage.Post` method to generate the messages you want to display. Each message specifies a severity and a timestamp. The message can also specify an icon, a category, and additional execution information. Double-click a message or right-click a message and select **Goto Location In»Step** from the context menu to go to the step that generates the message. By default, the sequence editor generates output messages for any information a source code control provider generates.



Note The Watch View and Output panes are available in the sequence editor but not in the TestStand User Interfaces.

Related concepts:

- [Sequence Context](#)

Terminating and Aborting Executions

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

When you terminate an execution, all the Cleanup steps, including subsequences the Cleanup steps call, execute normally and disregard the pending termination. If you

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

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

Related concepts:

- [TestStand ActiveX API Overview](#)
- [Checking for Suspended or Stopped Execution within Code Modules](#)

Checking for Suspended or Stopped Execution within Code Modules

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

In addition, you might want code modules to permit TestStand to suspend the parent execution thread at a breakpoint without requiring the code module to first return to TestStand. For example, you might want TestStand to be able to suspend an execution at a breakpoint while a code module launches a dialog box that waits for user input.

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

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

Because the **Allow Break While in Code Modules** option on the Execution tab of the Station Options dialog box is enabled by default, TestStand suspends the execution thread at a breakpoint even if other threads are executing code modules. Therefore, you do not need to use the Thread.ExternallySuspended property unless you disable the Allow Break While in Code Modules option.

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

LabVIEW

As an alternative to the Execution API, you can use the following VIs to enable VIs that TestStand calls to determine whether the execution receives a request to terminate or abort the execution:

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

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

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

- <TestStand Public>\Examples\Demos\Computer Motherboard Test\LabVIEW

Because the **Allow Break While in Code Modules** option on the Execution tab of the Station Options dialog box is enabled by default, TestStand suspends the execution thread at a breakpoint even if other threads are executing code modules. Therefore, you do not need to use the TestStand – Set Thread Externally Suspended VI unless you disable the Allow Break While in Code Modules option.

LabWindows/CVI

As an alternative to the Execution API, you can use the `TS_CancelDialogIfExecutionStops` function the TestStand Utility Functions driver provides to enable code modules that launch dialog boxes to verify whether the execution that called the function has stopped. Refer to the dialog box code in the following example source file for examples of how to use this function:

- `<TestStand Public>\Examples\Demos\Computer Motherboard Test\CVI\Computer Motherboard Test Sequence.c`

Related concepts:

- [TestStand Directory Structure](#)
- [TestStand Utility Functions Library](#)

Executing Sequences Remotely

When you use the Sequence Call Module tab to specify a sequence file pathname for a Sequence Call step and specify **Use Remote Computer** in the Execution Options section, TestStand locates the sequence file according to the type of path, as the following table describes.

Type of Path	Location at Edit Time	Location During Execution	Example
Relative	In the TestStand search paths you configure on the client (local) computer	In the TestStand search paths you configure on the server (remote) computer	<code>Transmit.seq</code>
Absolute	On the client (local) computer	On the server (remote) computer	<code>C:\Projects\Transmit.seq</code>
Network	On the computer the network path specifies	On the computer the network path specifies	<code>\\Remote\Projects\Transmit.seq</code>

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

client computer, TestStand resolves the path for the sequence file on the server computer.

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

- Add a common pathname to the search paths for the client and the server computers so that each resolves to the same relative pathname.
- Duplicate the files on the client and the server computers so that the file you edit on the client computer is identical to the file the server computer executes.
- Use absolute paths that specify a mapped network drive or full network path so that the file the client computer edits and the file the server computer executes are the same sequence file.

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

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



Note TestStand does not support remotely executing a sequence across different TestStand versions, for example, remotely executing a TestStand 2014 sequence from a computer running TestStand 2013.

Related concepts:

- [Search Directories](#)
- [TestStand ActiveX API Overview](#)
- [Configuring Remote Executions](#)

Configuring Remote Executions

You must properly configure a remote computer and the TestStand server application on the remote computer when you want to invoke a sequence on the remote computer from TestStand on a client computer. You must enable the TestStand server on the

remote computer to accept remote execution requests. You must also configure Windows system security to allow users to access and launch the TestStand server remotely. When you use the TestStand API on the server computer to access objects on the client computer, you must also configure Windows system security on the local client.



Note TestStand does not support remotely executing a sequence across different TestStand versions, for example, remotely executing a TestStand 2014 sequence from a computer running TestStand 2013.

Related concepts:

- [Configuring TestStand as a Server for Remote Executions](#)
- [TestStand ActiveX API Overview](#)
- [Executing Sequences Remotely](#)

Configuring TestStand as a Server for Remote Executions

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



Note TestStand does not support remotely executing a sequence across different TestStand versions, for example, remotely executing a TestStand 2014 sequence from a computer running TestStand 2013.

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

Enable the **Show the System Tray Icon While the TestStand Remote System is Active on this Machine** option on the Remote Execution tab of the Station Options dialog box

on the remote computer to make the TestStand icon visible in the remote computer system tray for each instance of the remote engine application. The tooltip for the icon indicates which computer is connected to the remote engine. Right-click the TestStand icon to display when the engine was created or to force the remote engine application to close.



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

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

Setting Windows System Security on a Remote Server

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

Related concepts:

- [DCOM Settings for Remote Servers](#)
- [Windows Firewall Settings for Remote Servers](#)

DCOM Settings for Remote Servers

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

(Windows 10/8.1/7) Windows launches a User Account Control elevation prompt for you to manually resolve when you enable this option.



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

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

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

In addition, when you enable the Allow All Users Access From Remote Machines option, the remote engine application overrides the application DCOM authentication level setting and does not authenticate connections. When you disable this option, TestStand revokes all the privileges listed previously, even if they were granted before the option was enabled.

When you do not enable the Allow All Users Access From Remote Machines option, complete the following steps to configure the DCOM security permissions for the server on a remote computer.

1. Log in as a user with administrator privileges.
2. Navigate to the standard Windows Control Panel facility for administrative tools and select **Component Services** or run `dcomcnfg` from the command line to launch the Component Services window.
3. On the left pane of the Component Services window, select **Component Services»Computers»My Computer**.
4. Right-click **My Computer** and select **Properties** to launch the My Computer Properties dialog box.

5. On the Default Properties tab of the My Computer Properties dialog box, enable the **Enable Distributed COM on this computer** option.



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

6. Click the **COM Security** tab of the My Computer Properties dialog box and complete the following steps to set permissions.
 - a. Click the **Edit Limits** button in the Access Permissions section to launch the Access Permission dialog box.
 - b. Click **Add** to add the users for whom you want to grant remote access. If the computer is not on a domain or the user is not a domain account, add the ANONYMOUS LOGON user.
 - c. (Windows 10/8.1/7) Click **OK** to close the Select Users or Groups dialog box.
 - d. Select the user you added and enable **Remote Access** in the Permissions section.
 - e. Click **OK** to close the Access Permission dialog box.
 - f. (Windows 10/8.1/7) Click the **Edit Limits** button in the Launch and Activation Permissions section to launch the Launch and Activation Permission dialog box.
 - g. Click **Add** to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the ANONYMOUS LOGON user.
 - h. (Windows 10/8.1/7) Click **OK** to close the Select Users or Groups dialog box.
 - i. Select the user you added and enable **Remote Launch and Remote Activation** in the Permissions section.
 - j. (Windows 10/8.1/7) Click **OK** to close the Launch and Activation Permission dialog box.



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

7. Click **OK** to close the My Computer Properties dialog box.
8. On the left pane of the Component Services window, select **My Computer»DCOM Config** to display a list of applications on the right pane.

9. Right-click **NI TestStand Remote Engine** and select **Properties** from the context menu to launch the NI TestStand Remote Engine Properties dialog box.
10. On the General tab of the NI TestStand Remote Engine Properties dialog box, set the **Authentication Level** to **None** if the user is not a domain account.
11. On the Identity tab of the NI TestStand Remote Engine Properties dialog box, select the **This user** option and enter a username and password or select the **The interactive user** option.
 - If you select the **This user** option, you must disable the **Show the System Tray Icon While the TestStand Remote System is Active on this Machine** option on the Remote Execution tab of the Station Options dialog box.
 - If you select the **The interactive user** option, a user must be logged in at the physical computer console during remote sequence execution. You can automatically log on a user each time a computer reboots by setting registry keys. Refer to Microsoft support article 315231 for more information about activating automatic logon.



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

12. Click the **Security** tab of the NI TestStand Remote Engine Properties dialog box and complete the following steps to set permissions.
 - a. (Windows 10/8.1/7) Select the **Customize** option and click the **Edit** button in the Launch and Activation Permissions section to launch the Launch and Activation Permission dialog box.
 - b. Click **Add** to add the users for whom you want to grant remote access. If the computer is not on a domain or the user is not a domain account, add the ANONYMOUS LOGON user.
 - c. (Windows 10/8.1/7) Click **OK** to close the Select Users or Groups dialog box.
 - d. Select the user you added and enable Remote Launch and Remote Activation in the Permissions section.
 - e. (Windows 10/8.1/7) Click **OK** to close the Launch and Activation Permission dialog box.
 - f. Select the **Customize** option and click the **Edit** button in the Access Permissions section to launch the Access Permission dialog box.
 - g. Click **Add** to add the users for whom you want to grant remote access. If the computer is not on a domain or the user is not a domain account, add the

ANONYMOUS LOGON user.

- h. (Windows 10/8.1/7) Click **OK** to close the Select Users or Groups dialog box.
 - i. Select the user you added and enable **Remote Access** in the Permissions section.
 - j. Click **OK** to close the Access Permission dialog box.
13. Click **OK** to close the NI TestStand Remote Engine Properties dialog box.

Related concepts:

- [Setting Windows System Security on a Remote Server](#)

Windows Firewall Settings for Remote Servers

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

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

1. Log in as a user with administrator privileges.
2. Navigate to the standard Windows Control Panel facility for Windows firewall and click **Allow an app or a program or feature through Windows Firewall** on the left panel of the Windows Firewall window to launch the Allowed Apps or Programs window.
3. Complete the following steps to add exceptions for the `REngine.exe` application with the firewall enabled.
 - a. Click the **Allow another app or program** button to launch the Add an App or Program dialog box.
 - b. Click Browse and select `<TestStand>\Bin\REngine.exe`.
 - c. Click **Add** to close the Add an App or Program dialog box.
 - d. Click **OK** to close the Allowed Apps or Programs window.
4. Click **Advanced settings** on the left panel of the Windows Firewall window to launch the Windows Firewall with Advanced Security window.
5. Complete the following steps to specify a security exception for DCOM.
 - a. Click **Inbound Rules** on the left panel of the Windows Firewall with Advanced

- Security window to display the list of inbound rules.
- Click **New Rule** on the Actions pane to launch the New Inbound Rule Wizard.
 - Select **Port** and click the **Next** button.
 - Select **TCP**, select **Specific local ports**, enter 135 in the Specific local ports control, and click **Next**.
 - Select Allow the connection and click Next.
 - Ensure that the **Domain**, **Private**, and **Public** options are enabled and click **Next**.
 - In the **Name** control, enter DCOM and click **Finish** to close the New Inbound Rule Wizard.

Related concepts:

- [Setting Windows System Security on a Remote Server](#)

Settings Windows System Security on a Local Client

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

Related concepts:

- [DCOM Settings for a Local Client](#)
- [Windows Firewall Settings for the Local Client](#)
- [Setting Windows System Security on a Remote Server](#)

DCOM Settings for a Local Client

Complete the following steps to configure DCOM access permissions on the client computer. You must restart TestStand for these changes to take effect.

1. Log in as a user with administrator privileges.
2. Navigate to the standard Windows Control Panel facility for administrative tools and select **Component Services** or run `dcomcnfg` from the command line to launch the Component Services window.
3. On the left pane of the Component Services window, select **Component Services»Computers»My Computer**.
4. Right-click **My Computer** and select **Properties** to launch the My Computer Properties dialog box.
5. Click the **COM Security** tab of the My Computer Properties dialog box and complete the following steps to set permissions:
 - a. Click the **Edit Limits** button in the Access Permissions section to launch the Access Permissions dialog box.
 - b. Select **ANONYMOUS LOGON** in the user name list and enable **Remote Access** in the Permissions Section. If ANONYMOUS LOGON does not appear in the user name list, click the **Add** button to add it.
 - c. Click **OK** to close the Access Permissions dialog box.

You must also disable DCOM authentication for the client application if it is not the TestStand Sequence Editor (`SeqEdit.exe`) or one of the TestStand User Interfaces named `testexec.exe`. For client applications built with LabVIEW, disable DCOM authentication for the application by adding the following line to the INI file associated with the application, as in `yourclient.ini` where `yourclient.exe` is the name of the application: `ole.AuthnLevel=1`

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

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

Windows Firewall Settings for the Local Client

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

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

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

Related concepts:

- [Settings Windows System Security on a Local Client](#)

Step Execution

Depending on the options you specify when you configure a step, a step performs multiple actions as it executes. The following table lists the most common actions a step can take, in the order the step performs them.

Action Number	Description	Remarks
1	Allocate step result	—
2	Enter batch synchronization section	When option is set
3	Check run mode for Skip	Sets the corresponding status on the step, then proceeds to Action Number 34
4	Evaluate precondition	When <code>False</code> , performs Action Number 29, then proceeds to Action Number 33
5	Acquire step lock	When option is set
6	Check run mode for Force Pass or Force Fail	Sets the corresponding status on the step, then proceeds to Action Number 30
7	Load module if not already loaded	—
8	Execute step switching	—
9	Evaluate Loop Initialization expression	Only when looping
10	Evaluate Loop While expression, skip to Action Number 27	Only when looping

Action Number	Description	Remarks
	when False	
11	Allocate loop iteration result	Only when looping
12	Call Pre-Step Engine callbacks	—
13	Evaluate Pre-Expression	—
14	Call Pre-Step substeps for step type	—
15	Process additional results for input parameters	—
16	Call module	—
17	Process additional results for output parameters	—
18	Call Post-Step substeps for step type	TestStand calls Post-Step substeps even when the user code module generates a run-time error, which enables Post-Step substeps to perform error handling, when appropriate. If Post-Step substeps clear run-time errors, proceed to Action Number 19. If step is configured to ignore run-time errors, proceed to Action Number 23.
19	Evaluate Post-Expression	—
20	Evaluate Status expression	—
21	Call Post-Step Engine callbacks	—
22	Call Post-Step	Only when loop iteration fails

Action Number	Description	Remarks
	Failure Engine callback	
23	Populate loop iteration result	Only when looping
24	Call Post-ResultList Entry Engine callback	Only when looping
25	Call Post-Results Engine callback	Only when looping
26	Evaluate Loop Increment expression, return to Action Number 10	Only when looping
27	Evaluate Loop Status expression	Only when looping
28	Disconnect switching routes with step lifetime	—
29	Unload module when required	—
30	Update sequence failed state	—
31	Call Post-Step Failure Engine callback	Only when step fails
32	Execute post action	—
33	Release step lock	When option is set

Action Number	Description	Remarks
34	Exit batch synchronization section	When option is set
35	Populate step result	Custom additional results also populate at this time. However, additional results for input and output parameters populate immediately before and after Action Number 16, respectively.
36	Call Post-ResultList Entry Engine callback	—
37	Call Post-Results Engine callback	—

Usually, a step performs only a subset of these actions depending on the configuration of the step and the test station. When TestStand detects a run-time error in one of these actions, it calls the Post-Step Error Engine callbacks. When you do not define these callbacks or when the callbacks do not reset the error state for the step, TestStand executes the following actions to clean up the execution of the step:

- If the error occurs before the step calls the code module, TestStand does not call the code module and proceeds to Action Number 28.
- If the error occurs while the step is calling the code module, TestStand proceeds to Action Number 28.
- If the error occurs after the step calls the code module, TestStand calls Action Number 21, if it is not already called, and proceeds to Action Number 28.
- If a run-time error occurs in a loop iteration, TestStand also populates the loop iteration result, Action Number 23.

Related concepts:

- [Updating the Status Bar Using UI Messages](#)

Result Collection

TestStand automatically collects the results of each step. You can configure result

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

You can generate a report and log results to a database to process the collected test results.

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

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

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

Report Generation

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

Related concepts:

- [Database Logging](#)
- [Custom Result Properties](#)
- [Exceptions to Custom Result Properties](#)
- [Standard Result Properties](#)

- [Loop Results](#)

Related information:

- [Locals Context Menu - Sequence File Tab](#)

Standard Result Properties

In addition to copying step properties to step results, TestStand also adds a set of standard properties to each step result as subproperties of the `Step.Result.TS` property, as described in the following table.

Table 11. Standard Result Properties

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

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

Related concepts:

- [Result Collection](#)
- [Custom Result Properties](#)

Custom Result Properties

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

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

- **Step.Result.Error.Code**—Code that describes the error that occurred.



Note TestStand supports only 32-bit signed integer values for the run-time error code property. Although you can assign a larger or fractional value to the property, TestStand will truncate the value to 32-bit when handling and propagating a run-time error.

- **Step.Result.Error.Msg**—Message string that describes the error that occurred.

- **Step.Result.Error.Occurred**—Boolean flag that indicates a run-time error occurred in the step. TestStand documentation refers to this property as the error occurred flag. The error occurred flag can become `True` when a run-time error condition occurs and the code module or module adapter sets the value to `True` or when an unhandled exception occurs in the code module or at any other time during step execution. When a step finishes execution and the error occurred flag is `True`, the TestStand Engine responds in the following ways:
 - Does not evaluate the status and post-expressions for a step and sets the step status to `Error`.
 - Evaluates the Ignore Run-Time Errors step property.
 - When this property is `False`, TestStand reports the run-time error to the sequence.
 - When this property is `True`, TestStand continues execution normally after the step.
- **Step.Result.Status**—The status of the last execution of the step, such as `Done`, `Passed`, `Failed`, `Skipped`, or `Error`. TestStand documentation refers to this property as the step status. Before TestStand executes a step, it sets the step status to `Running` or `Looping`. When a step finishes execution and the error occurred flag is `False`, TestStand changes the step status to `Done`. The step status becomes `Passed` or `Failed` only when a code module, module adapter, or step type explicitly sets the step status to `Passed` or `Failed`.
- **Step.Result.ReportText**—Message string TestStand includes in the report.
- **Step.Result.Common**—Placeholder container you can customize by modifying the `CommonResults` standard data type.



Note The preceding list does not include the result properties that are unique for each built-in step type or for Synchronization, Database, Interchangeable Virtual Instrument (IVI), or LabVIEW Utility step types.

The `Common` property uses the `CommonResults` custom data type and is a subproperty of the `Result` property for every step type. Consequently, you can add a subproperty to the result of every step type by adding a subproperty to the definition of the `CommonResults` custom data type. When you modify the `CommonResults` data type, you must ensure that the new type definition does not conflict with earlier versions of the type in other files. NI recommends modifying the `CommonResults` data type only when you want to perform an architectural change to all step types that you

use.



Note Share the modified `CommonResults` data type and the step types that use the `CommonResults` data type only with systems on which you are certain you will not deploy any conflicting changes to `CommonResults`.

Related concepts:

- [Error and CommonResults](#)
- [Built-In Step Types](#)
- [Type Versioning](#)
- [Modifying Types](#)
- [Result Collection](#)
- [Exceptions to Custom Result Properties](#)
- [Standard Result Properties](#)

Exceptions to Custom Result Properties

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

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

Related concepts:

- [TestStand ActiveX API Overview](#)
- [Step Types That Work with Specific Module Adapters](#)
- [Custom Result Properties](#)
- [Result Collection](#)

Subsequence Results

When a step calls a subsequence or generates a call to a callback sequence, TestStand creates a special property result of the `Step.Result.TS` property to store the result of the subsequence unless the callback or sequence disables results. The following table lists the name of the subproperty for each type of subsequence call.

Result Subproperty Name	Type of Subsequence Call
<code>TS.SequenceCall</code>	Sequence Call
<code>TS.PostAction</code>	Post Action callback
<code>TS.SequenceFilePreStep</code>	SequenceFilePreStep callback
<code>TS.SequenceFilePostStep</code>	SequenceFilePostStep callback
<code>TS.ProcessModelPreStep</code>	ProcessModelPreStep callback
<code>TS.ProcessModelPostStep</code>	ProcessModelPostStep callback
<code>TS.StationPreStep</code>	StationPreStep callback
<code>TS.StationPostStep</code>	StationPostStep callback
<code>TS.SequenceFilePreInteractive</code>	SequenceFilePreInteractive callback
<code>TS.SequenceFilePostInteractive</code>	SequenceFilePostInteractive callback
<code>TS.ProcessModelPreInteractive</code>	ProcessModelPreInteractive callback
<code>TS.ProcessModelPostInteractive</code>	ProcessModelPostInteractive callback
<code>TS.StationPreInteractive</code>	StationPreInteractive callback
<code>TS.StationPostInteractive</code>	StationPostInteractive callback
<code>TS.SequenceFilePostResultListEntry</code>	SequenceFilePostResultListEntry callback
<code>TS.ProcessModelPostResultListEntry</code>	ProcessModelPostResultListEntry callback
<code>TS.StationPostResultListEntry</code>	StationPostResultListEntry callback
<code>TS.SequenceFilePostStepRuntimeError</code>	SequenceFilePostStepRuntimeError callback
<code>TS.ProcessModelPostStepRuntimeError</code>	ProcessModelPostStepRuntimeError callback
<code>TS.StationPostStepRuntimeError</code>	StationPostStepRuntimeError callback
<code>TS.SequenceFilePostStepFailure</code>	SequenceFilePostFailure callback

Result Subproperty Name	Type of Subsequence Call
<code>TS.ProcessModelPostStepFailure</code>	ProcessModelPostStepFailure callback
<code>TS.StationPostStepFailure</code>	StationPostStepFailure callback

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

- **ThreadID**—ID of the thread used to execute the sequence.
- **SequenceFile**—Absolute path of the sequence file that contains the subsequence.
- **SequenceFileVersion**—Version of the sequence file that contains the subsequence.
- **Sequence**—Name of the subsequence the step called.
- **Status**—Status of the subsequence the step called.
- **ResultList**—Value of `Locals.ResultList` for the subsequence the step called. This property contains the results for the steps in the subsequence.

Related concepts:

- [Callback Sequences](#)
- [List of Engine Callbacks](#)
- [Result Collection](#)
- [Custom Result Properties](#)

Loop Results

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

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

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

Engine Callbacks

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

Use Engine callbacks to direct 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.

Related concepts:

- [List of Engine Callbacks](#)
- [Interactively Executing Steps](#)
- [Modifying Engine Callbacks](#)

Step Status

Every step has a `ResultStatus` property, which is a string constant that indicates the result of the step execution. Although TestStand imposes no restrictions on the values to which the step or the code module of the step can set the status property, TestStand and the built-in step types use and recognize the values that appear in the following table.

String Value	Meaning	Source of the Status Value
Passed	Indicates the step performed a test that passed, or the step did not execute because the run mode for the step is Force Pass.	Step, code module, or TestStand
Failed	Indicates the step performed a test that failed, or the step did not execute because the run mode for the step is Force Fail.	Step, code module, or

String Value	Meaning	Source of the Status Value
	When you enable the Step Failure Causes Sequence Failure option on the Run Options panel of the Step Settings pane, TestStand sets the sequence status to <code>Failed</code> when a step fails. When the sequence returns as <code>Failed</code>, the Sequence Call step also fails. In this way, a step failure in a subsequence can propagate up through the chain of Sequence Call steps. By default, TestStand enables the Step Failure Causes Sequence Failure option for most step types. You can also control how execution proceeds after a step failure causes a sequence to fail. To configure an execution to jump to the Cleanup step group upon failure, enable the Immediately Goto Cleanup on Sequence Failure option in the Sequence Properties dialog box. By default, TestStand disables this option.	TestStand
Error	Indicates a run-time error occurred.	TestStand
Done	Indicates the step completed without setting the status.	TestStand
Terminated	Indicates the step did not set the status and a request to terminate the execution occurred while executing the step. When you request to terminate an execution, the currently executing sequence invocation for each thread waits for the current running step to complete before initiating termination. A terminating sequence runs steps in the Cleanup step group, if not already executing, and returns to the calling sequence. When a terminating subsequence returns to a calling sequence, TestStand sets the calling sequence step status to <code>Terminated</code>, and the calling sequence continues the termination process up the call stack unless you enabled the Ignore Termination option on the Run Options panel of the Step Settings pane for the Sequence Call step. When you enable this setting, TestStand ignores the termination of the execution for the thread, and the thread execution continues normally. If TestStand returns the request to terminate the execution to the root sequence invocation, the result status for the execution is <code>Terminated</code>.	TestStand
Skipped	Indicates the step did not execute because the run mode for the step is Skip.	TestStand

String Value	Meaning	Source of the Status Value
Running	Indicates the step is currently running.	TestStand
Looping	Indicates the step is currently running in loop mode.	TestStand

Failures

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

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

Terminations

When you request to terminate an execution, the currently executing sequence invocation for each thread waits for the current running step to complete before initiating termination. A terminating sequence runs steps in the Cleanup step group, if not already executing, and returns to the calling sequence. When a terminating subsequence returns to a calling sequence, TestStand sets the calling sequence step status to `Terminated`, and the calling sequence continues the termination process up the call stack unless you enabled the Ignore Termination option on the Run Options panel of the Step Settings pane for the Sequence Call step. When you enable this setting, TestStand ignores the termination of the execution for the thread, and the thread execution continues normally. If TestStand returns the request to terminate the execution to the root sequence invocation, the result status for the execution is `Terminated`.

Related concepts:

- [Built-In Step Types](#)

Run-Time Errors

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

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

When a run-time error occurs, the sequence runs steps in the Cleanup step group, if not already executing, and returns to the calling sequence. When a subsequence with a run-time error returns to a calling sequence, TestStand sets the calling sequence step status to `Error`, and the calling sequence continues to propagate the run-time error up the call stack unless you enabled the **Ignore Run-Time Errors** option on the Run Options panel of the Step Settings pane for the Sequence Call step. When you enable this setting, TestStand ignores the error for the thread, and the thread execution continues normally. If TestStand returns the run-time error to the root sequence invocation, the result status for the execution is `Error`.

To interactively handle run-time errors, configure TestStand to launch the Run-Time Error dialog box in the event of an error by selecting **Show Dialog** from the On Run-Time Error ring control on the Execution tab of the Station Options dialog box.

You can also invoke Post-Step RunTime Error Engine callbacks when a run-time error occurs.

Related concepts:

- [List of Engine Callbacks](#)

Using TestStand on SMP Systems

Symmetric multiprocessing (SMP) systems are computers that contain multiple CPUs, where a CPU can be a single core on a multi-core processor or a single-core processor on a multiprocessor computer. Microsoft Windows and other modern operating systems can take advantage of SMP systems to achieve increased performance. SMP systems achieve better performance by executing multiple threads on multiple CPUs concurrently. Performance improvement can come through executing multiple processes concurrently as well as from executing multiple threads within a process concurrently. Generally, when an application is implemented with multiple threads, the operating system attempts to schedule each thread on a separate CPU when possible. In the best-case scenario, an SMP system can execute a multithreaded application n times faster than a non-SMP system, where n is the number of CPUs. An application that exhibits this best-case performance scaling is said to scale linearly with the number of CPUs.

An application scales linearly if the work done in the application is completely divided into an independent part for each CPU, with no data shared between the independent parts. If the parts have to interact with each other or access the same data, the performance increase can be substantially less than linear. In some cases, performance of an application can even be worse on an SMP system than on a non-SMP system. The performance degradation or lack of improvement on SMP systems is caused by a variety of factors. The most typical factors are data sharing, where both reads and writes occur to the same or nearby data in memory, and synchronization interdependencies, where threads must coordinate with each other for the application to execute correctly. The degree to which these factors impact performance is dependent on the particular CPU architecture in the SMP system. For example, on some quad-core CPUs the memory cache is shared among all of the cores while on others the cache is separate for each core. Such a difference can result in performance differences when the threads access a large amount of shared data.

General Strategies for Optimizing Performance on SMP Systems

The best way to optimize an application for an SMP system is to break up the application into as many independent, concurrent tasks as possible and run at least n of them at a time, in n separate threads, where n is the number of CPUs on the computer. The application should avoid sharing data among threads and avoid

requiring synchronization among threads as much as possible. Unfortunately, it is often difficult or impossible to design application functionality in a way that does not require threads to share data and does not require thread synchronization.

Because the performance impact of sharing data and synchronization depends on the CPU architecture, you can benchmark the application on different computer systems to determine the computer architecture on which the application performs best. You can also optimize the application for a particular computer system. If parts of the application perform worse with multiple CPUs, you can tune the application performance for a particular computer system by limiting the CPUs which certain threads or processes use to a subset of the CPUs on the system. You can then benchmark to determine how such tuning affects performance.

Strategies for Optimizing TestStand Performance on SMP Systems

TestStand applications are often multithreaded. You can execute sequences in parallel by using a Sequence Call step to execute a sequence in a new thread or new execution. You can execute code modules in parallel by executing the sequences which contain them in parallel. You can also create threads in code modules using your programming environment.

TestStand sequences that execute in parallel threads, communicate with the engine, share data with other sequences, and synchronize with other sequences might not scale well on SMP systems. Additionally, code modules that sequences call might also perform operations that do not scale well on SMP systems. Therefore, your system might have better performance if you constrain some or all threads to a subset of CPUs. If the TestStand sequences spend significant time in code modules that are completely independent, the system may have the best performance by using all CPUs for such code modules while constraining the CPUs the rest of the system uses.

To allow you to tune your system to achieve the best performance, TestStand provides several features to control how the TestStand process and its threads utilize the CPUs on your computer. In the descriptions of the features below, the CPU affinity of a process refers to the set of CPUs on which the threads in the process may run. The CPU affinity of a thread refers to the subset of those CPUs on which a particular thread may run.

- **Station Option: Default CPU Affinity for Threads**—Determines the default CPU

affinity for new threads of executions and for the user interface thread. The default is to allow all CPUs. This option is located on the Preferences tab of the Station Options dialog box.

- **Sequence Call Step: CPU Affinity for New Thread or Execution**—The CPU affinity of the new thread in the current execution or the initial thread in a new execution. By default the new thread uses the Default CPU Affinity for Threads station option. This setting allows you to specify that the new thread allows all CPUs, uses the CPU affinity of the calling sequence, or allows specific CPUs that you specify. This option is located on the Sequence Call Advanced Settings window.
- **CPU Affinity Step**—Enables you to get or set the CPU affinity of the process or the current execution thread. This step gives you low-level control over how your system utilizes CPUs. This step is located in the Advanced subgroup of the Synchronization group on the Insertion Palette or in the Insert Step submenu of the Steps pane context menu.



Note TestStand represents CPU affinity as an integer number where each bit in the binary representation specifies a CPU, where a CPU can be a single-core processor or a single core on a multi-core processor. The lowest ordered bit specifies the first CPU. When specifying a CPU affinity you may want to use a binary number. You specify a binary number in TestStand with the `0b` prefix. For example, setting the CPU affinity for a thread to `0b1010` specifies to allow the thread to execute on the second or fourth CPU. You can use `-1` for the CPU affinity of the thread to specify that the thread may run on any CPU on which the process may run. You can use `-1` for the CPU affinity for the process to specify that the process may run on any CPU on the computer.

You can use the CPU affinity features together to tune your test system to get the best performance on a particular computer. For example, if you have a quad core computer, you could set the Default CPU Affinity for Threads Station Option to `0b0011` to restrict the test system to use only the first two CPUs and benchmark to determine how the performance of the system is affected. If parts of the test system contain code modules that can be highly parallelized and that do a significant amount of work, you could then use the CPU Affinity settings of a Sequence Call step or a CPU Affinity step to set the execution threads to allow all CPUs for such parts of the system. You may try various CPU affinity configurations to determine what gives the test system the best performance on a particular computer. Since performance results depend on CPU

architecture, the best tuning for one computer may be different from the best tuning on another computer.

Related concepts:

- [Configuring the CPU Affinity for LabVIEW Execution Threads](#)
- [Symmetric Multiprocessing in VIs Executed from TestStand](#)

Built-In Step Types

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

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



Note TestStand also includes application-specific step types. For example, TestStand provides sets of step types to more easily synchronize multiple threads, access databases, control modular instruments, control I/O instruments, and access VIs and remote systems.

Related concepts:

- [Step Types You Can Use with Any Module Adapter](#)
- [Step Types That Work with Specific Module Adapters](#)
- [Step Types That Do Not Use Module Adapters](#)
- [Instrument Control Step Types](#)

Creating Steps from Step Types

Use step types when you insert steps in the Setup, Main, and Cleanup groups of the Steps pane in the Sequence File window. You can insert a step using the Step Types list in the Insertion Palette or the Insert Step submenu in the Steps pane context menu. The Insertion Palette and the Insert Step submenu list all the available step types.

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

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

To modify step properties common to all step types, click the Properties tab of the Step Settings pane.

The Insertion Palette also contains a Templates list you can use to hold copies of sequences, steps, and variables you reuse.

Related concepts:

- [Built-In Step Properties](#)

Step Types You Can Use with Any Module Adapter

You can use the following built-in step types with any module adapter:

- Pass/Fail Test
- Numeric Limit Test

- Multiple Numeric Limit Test
- String Value Test
- Action

When you insert a step in a sequence, you must select a module adapter in the Step Types list of the Insertion Palette or from the Adapter ring control on the TestStand Sequence Editor toolbar. TestStand assigns the adapter you selected when you insert the step. Once you add a step, you can change the adapter associated with the selected step on the General panel on the Step Settings pane.



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

When you select the **<None>** adapter, the step does not call a code module.

To specify the code module the step calls, select **Specify Module** from the step context menu or click the Module tab on the Step Settings pane.

Related concepts:

- [Step Types That Work with Specific Module Adapters](#)

Step Types That Work with Specific Module Adapters

The following step types use specific module adapters. When you create an instance of one of these step types, use the edit tabs or dialog boxes, which you access through the context menu of the step or the Step Settings pane, to configure the step.

- Sequence Call
- Run VI Asynchronously

Step Types That Do Not Use Module Adapters

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

- Flow Control
- Statement
- Label
- Message Popup
- Call Executable
- Property Loader
- Legacy Property Loader
- FTP Files
- Additional Results
- Synchronization Step Types
- Database Step Types
- IVI Step Types
- LabVIEW Utility Step Types



Note The ICMP package must be installed to use IVI Step Types. These steps are 32-bit only.

Using the Property Loader Step Type

Refer to the following topics for information on using the Property Loader step type.

Importing and Exporting with a Legacy Source

Importing and Exporting with a Legacy Source

To use the Legacy Import/Export Properties tool in TestStand, you must add it to the Tools Menu. Complete the following steps to add the Legacy Import/Export Properties tool to the Tools Menu:

1. Copy `<TestStand>\Components\StepTypes\Database\LegacyImportExportTool.ini` to the `<TestStandPublic>\Setup\ToolMenusToInstall` directory.
2. Restart TestStand.

Use the Legacy Import/Export Properties tool from the Tools Menu to export data to legacy formats and to import data from legacy formats.

To use legacy source with the new Property Loader step, create an instance of the property loader step in your sequence. In the Source Type, select one of the following legacy source types:

- NI Legacy Tab Delimited Text (*.txt)
- NI Legacy Comma Delimited Text (*.csv)
- NI Legacy Excel File (*.xls)
- NI Legacy Database

The following features are not supported when using legacy sources in the new Property Loader step:

- Using alias names
- Import status in `Step.Result`
- Import Sequence File Attributes, Sequence Attributes, Step Attributes and Sequence Parameters
- Property Loader groups
- Override default column names for selected properties in database



Note You need to use plugin-specific options to configure options like Start and End markers for files, filtering for databases, etc. For more information about plugin-specific import and export options, refer to the help topic for the plugin you want to use.

Importing Data to Non-Existing Properties

Importing Data to Non-Existing Properties

You can use the Property Selector step to import to a property which will be created

during runtime or you can configure the Property Loader step to create non-existing properties.

To import values to the non-existing properties, you can choose to perform one of the following options:

Option 1: Make sure the property loader source contains the details about the non-existing property and choose **Import All Data from Source** option.

Option 2: Make sure the property loader source has a group named Expressions that contains the details about the non-existing properties. Use the Expressions group in the property selector to specify the lookup of non-existing properties. The value of the expression should be as follows:

- `StationGlobals.<Lookup>` - Specifies station global variable to import
- `FileGlobals.<Lookup>` - Specifies file global variable to import
- `Sequence[ '<SequenceName>' ].Locals.<Lookup>` - Specifies sequence local to import
- `Sequence[ '<SequenceName>' ].Parameters.<Lookup>` - Specifies sequence parameter to import
- `Sequence[ '<SequenceName>' ].Step[ '<UniqueStepId>' ].<Lookup>` - Specifies the step variable to import

`<Lookup>` - Replace it with the lookup of the variable to import

`<SequenceName>` - Replace it with the name of the sequence under which the variable exists to import

`<UniqueStepId>` - Replace it with the unique step id representing the step. Unique step id should be calculated based on what is configured for Unique Step Id in the property loader configuration.

Validate and Verify Runtime Behavior of Import

You can run the analyzer to validate the settings configured by the user in property loader step and also to validate the property loader source.

You can use Import Preview, which launches the Import Preview Dialog, displaying the

state of the sequence file after importing. It also shows what properties are updated, along with the old and new values.

You can choose to log the lookup of the updated property and its old and new value in the property loader step. After executing the step, you can verify the updated properties and their values by looking at the logged results.

Using Alias Names in Property Loader

Using Alias Names in Property Loader

Alias names are alternate names used to represent a property in the property loader source. You need not use property lookup in the property loader source if you use alias names and, therefore, you can create a property loader source that does not contain any TestStand-specific terminology (i.e. Locals, FileGlobals, etc).

If your property loader source is a file, the mapping between alias names and property lookup should exist in a file with extension `.pla`. If your property loader source is a database, the mapping between alias names and property lookup should exist in a table in the database. You can provide the filepath or table name to specify where the mapping between alias names and property lookup exists.

When exporting with the Import/Export Properties tool, you can use the property selector to specify alias names for the property.

Alias Name Restrictions

Alias names cannot contain the following characters:

`. < > [ ] { } ' "`

Storing Alias Names in a File

Mapping between alias names and property lookup must be stored in a file with extension `.pla`. Review the example file located at `<TestStandPublic>\Examples \Built-In Step Types\Property Loader Step Type\AliasNames.pla` to see how alias names are stored in a file.

Storing Alias Names in a Database

Use a database table to store alias names and property lookup mappings. The Alias table schema contains the following columns:

ALIAS_NAME — Contains the name of the alias. This is the primary column for the table. Use the following data types for each database:

- Access – VARCHAR(255)
- MySQL – VARCHAR(255)
- SQLServer – VARCHAR(255)
- Sybase – VARCHAR(255)
- Oracle – VARCHAR2(255)

PROPERTY_LOOKUP — Contains the property lookup mapped to the specified alias name. Use the following data types for each database:

- Access – LONGTEXT
- MySQL – VARCHAR(8000)
- SQLServer – VARCHAR(8000)
- Sybase – VARCHAR(32767)
- Oracle – VARCHAR2(4000)



Note A property loader plugin can override where and how the alias names are stored. A plugin can choose to store the alias name information in the property loader source itself so that you don't have to remember to deploy the alias file.

Property Loader Plugins - Database Table Schemas

Database Table Schemas

The Property Loader step type includes a plugin to support importing and exporting properties to and from a database. This topic describes the table schemas that the database plugin requires and creates when exporting properties.



Note

- The database plugin is responsible for establishing a connection with the database. There is no such dependency on the step, unlike with the Legacy Property Loader step.
- The Property Loader step also supports importing and exporting limits from .csv, .txt, and .xlsx (Microsoft Excel) file types. Refer to the Property Loader Plugins - File Formats topic for more information about storing limits in these files.
- See the Legacy Property Loader Step - Loading Limit Values from Databases topic for more information about database table schemas for the Legacy Property Loader step.

GROUPS_DESC Table Structure

The following table describes the schema that defines the property groups, their names, and their values. Some columns are optional and can be null, meaning that they do not have to contain values.

Column Name	Data Type	Can Be Null?	Purpose
GROUP_ID	Primary Key, String	No	Unique value that identifies each entry in the table.
NAME	String	Yes	Stores the name of the property groups.
DESCRIPTION	String	Yes	Stores a description of the properties.

See the following image for an example of a GROUP_DESC table. Each row represents a different property group. In this case, there is only one group, called `New Properties`.

GROUP_ID	NAME	DESCRIPTION
8a612a9c-d71e-4c2e-bdc0-4be2ffd3e26f	New Properties	This is a description

PROPERTIES Table Structure

The following table describes the schema that defines the columns used by the Property Loader. Some columns are optional and can be null, meaning that they do not have to contain values.

Column Name	Data Type	Can Be Null?	Purpose
ID	Primary Key, Number	No	Unique value that identifies each entry in the table
GROUP_ID	Foreign Key, String	No	The corresponding GROUP_ID column of the GROUP_DESC table.
SEQUENCE_FILE_NAME	String	Yes	Indicates the name of the sequence file in which the property exists.
SEQUENCE_NAME	String	Yes	Indicates the name of the sequence that each property will be imported into. If the sequence name for a property is blank, the property will be imported into all sequences within the sequence file.
CATEGORY	String	No	Indicates the location of each property. The location can be one of the following literal values. • <code>{StationGlobals}</code>—Indicates that the property is a station global variable. • <code>{FileGlobals}</code>—Indicates that the property is a file global variable. • <code>{Locals}</code>—Indicates that the property is a local variable within a sequence. • <code>{Parameters}</code>—Indicates that the property is a sequence parameter. • <code>{Attributes}</code>—Indicates that the property is a sequence attribute. If the category is not one of the above values, CATEGORY must specify a value that can uniquely identify the step property. Typically, that is either the name or ID of a step, and the columns to the right specify the property of the

Column Name	Data Type	Can Be Null?	Purpose
			step. See the example PROPERTIES table image below for more information about step name and ID formats.
PROPERTY_LOOKUP	String	No	Indicates the lookup string of the property whose value will be updated. If the lookup is specified using an alias, the alias name is represented within <>. Refer to Rows 2, 3, 5, and 6 in the example image below for an example of alias name representation. If the lookup is for a container, the name of the container is enclosed in [] and suffixed with .Type. The VALUE column is null in this case. Refer to column 13 in the example image below for an example of array representation. If the database was exported with type information, values in this column will also contain the type information after the lookup string, enclosed within { }.
VALUE	String	Yes	Indicates the value of the property to be imported.

See the following image for an example of a PROPERTIES table. When the PROPERTIES table contains data associated with step properties, either the step name, step ID, or both should be specified in the CATEGORY column. Rows 1-6 demonstrate valid ways to specify these values.

ID	GROUP_ID	SEQUENCE_FILE_NAME	SEQUENCE_NAME	CATEGORY	PROPERTY_LOOKUP	VALUE
1	8a612a9c...	Property Loader - Loa...	MainSequence	Power On	Limits {Container}	
2	8a612a9c...	Property Loader - Loa...	MainSequence	Step[ID#:uYhBnrEX5hGzYpiQlqYQ+D]	<PowerOnLimitLow> {Number}	9
3	8a612a9c...	Property Loader - Loa...	MainSequence	Step["Power On"]	<PowerOnLimitHigh> {Number}	15
4	8a612a9c...	Property Loader - Loa...	MainSequence	Video	Limits {Container}	
5	8a612a9c...	Property Loader - Loa...	MainSequence	Step[ID#:vIhBnrEX5hGzYpiQlqYQ+D]	<VideoLimitLow> {Number}	70
6	8a612a9c...	Property Loader - Loa...	MainSequence	Step["Video"]	<VideoLimitHigh> {Number}	75
7	8a612a9c...	Property Loader - Loa...		{FileGlobals}	TempSweep {Number [0] .. [4]}	
8	8a612a9c...	Property Loader - Loa...		{FileGlobals}	TempSweep[0]	20
9	8a612a9c...	Property Loader - Loa...		{FileGlobals}	TempSweep[1]	40
10	8a612a9c...	Property Loader - Loa...		{FileGlobals}	TempSweep[2]	60
11	8a612a9c...	Property Loader - Loa...		{FileGlobals}	TempSweep[3]	80
12	8a612a9c...	Property Loader - Loa...		{FileGlobals}	TempSweep[4]	100
13	8a612a9c...	Property Loader - Loa...	MainSequence	{Locals}	CustomType {MyCustomType}	
14	8a612a9c...	Property Loader - Loa...	MainSequence	{Locals}	CustomType.MyNumber {Number}	0
15	8a612a9c...	Property Loader - Loa...	MainSequence	{Locals}	CustomType.MyString {String}	TestStand
16	8a612a9c...	Property Loader - Loa...	MainSequence	{Locals}	CustomType.MyBool {Boolean}	False

ALIAS Table Structure

The following table describes the schema that defines alias names for the Property

Loader step. The Property Loader step allows you to optionally specify alias names for properties within the database table. Aliases allow you to refer to properties within the database with meaningful names, rather than a property lookup string. When a PROPERTIES file contains aliases, a corresponding ALIAS table must be associated with the PROPERTIES table.



Note Use of an ALIAS table is optional.

Column Name	Data Type	Can Be Null?	Purpose
ALIAS_NAME	Primary Key, String	No	The alias name.
PROPERTY_LOOKUP	String	No	The lookup string that maps the alias name to the property.

See the following image for an example of an ALIAS table that shows the mapping between alias names and their lookup strings. Note that the alias name `PowerOnLimitLow` is used in the PROPERTIES table example above.

ALIAS_NAME	PROPERTY_LOOKUP
▶ PowerOnLimitLow	Sequence[MainSequence].Step[Power On].Limits.Low
PowerOnLimitHigh	Sequence[MainSequence].Step[Power On].Limits.High
VideoLimitLow	Sequence[MainSequence].Step[Video].Limits.Low
VideoLimitHigh	Sequence[MainSequence].Step[Video].Limits.High

Property Loader Plugins - File Formats

File Formats

The Property Loader step type and the Import/Export Properties tool includes file plugins to support .csv, .txt, and .xlsx (Microsoft Excel) file types. This topic describes the required delimiter characters, file structure, special property formats, and property aliases for these file formats.



Note

- The Property Loader step also supports importing and exporting properties from a database. Refer to the Property Loader Plugins - Database Format topic for more information about storing properties in a database.
- See the Legacy Property Loader Step - Loading Limit Values from Files topic for more information about different file formats for the Legacy Property Loader step.

Delimiter Characters

Column Delimiters

The primary difference between the file types is the delimiter character used to separate columns of data. The following table describes the column delimiter used for each file format:

File Format	Column Delimiter
.csv	Comma
.txt	Tab character
.xlsx	None. <code>.xlsx</code> files should be opened and edited with Microsoft Excel or another application capable of supporting <code>.xlsx</code> files.

Row Termination Characters

In all file formats, new-line characters are used to terminate each row of data.

Field Delimiters

Data values, or fields, within the file may optionally be enclosed in double quotes. However, when certain characters are present in a field, the field must be enclosed in double quotes. The following table describes the cases in which fields must be enclosed in double quotes:

File Format	Cases Where Field Must Be Enclosed in Double Quotes
.csv	- Fields containing commas - Fields containing leading or trailing spaces - Fields containing line breaks - Fields containing double quotes. When a field contains double quotes, each double quote must be preceded by another double quote to serve as an escape character.
.txt	- Fields containing tab characters - Fields containing leading or trailing spaces - Fields containing line breaks - Fields containing double quotes. When a field contains double quotes, each double quote must be preceded by another double quote to serve as an escape character.
.xlsx	None. It is not necessary to enclose fields in double quotes in .xlsx files.

File Structure

See the following image for an example of a Property Loader source file in the correct structure for use with the Property Loader plugins.



Note The file shown is in the Microsoft Excel format, but the same file structure applies to all file formats.

	A	B	C	D	E
1	<PropertyGroup name='PumpTestLimits' id='7771d6b1-30bd-4281-9e26-8360dafdd10b'>				
2	<Sequence>	<Category>	<PropertyLookup>	<Value>	
3	MainSequence	{Locals}	StartingVoltage {Number}	12	
4	MainSequence	Pump Test	Limits {Container}	Low {Number}	High {Number}
5	MainSequence	Pump Test	{ContainerValues}	12.5	25
6	</PropertyGroup>				

The purpose of each element in the file is described in the following table:

File Element	Position in Example File	Purpose
<PropertyGroup> start and end markers	Cells A1, A6	Indicate the beginning and end of the TestStand property data to be imported from the file. The <PropertyGroup> start marker contains the name and ID of the data, and should be unique within the file. Optionally, a description can be included in the <PropertyGroup> start tag. Any data that is not enclosed within the <PropertyGroup> start and end tags is ignored.
<Sequence> column header	Cell A2	Indicates that this column represents the name of the sequence that each property will be imported into. If the sequence name for a property is blank, the property will be imported into all sequences. Note: This column is optional. If the <Sequence> column does not exist in the file, all Locals, Parameters, Sequence Attributes, and step properties will be imported into all sequences within the sequence file.
<Category> column header	Cell B2	Indicates that this column represents the location of each property. The location can be one of the following literal values: {StationGlobals}—Indicates that the property is a station global variable. {FileGlobals}—Indicates that the property is a file global variable. {Locals}—Indicates that the property is a local variable within a sequence. {Parameters}—Indicates that the property is a sequence parameter. {Attributes}—Indicates that the property is a sequence attribute. If the name defined in the <PropertyGroup> start tag is GlobalProperties, the property is instead a sequence file attribute. If the category is not one of the above values, the category must specify a value that can uniquely identify the step property. Typically, that is either the name or ID of a step, and the columns to the right specify the property of the step. See the Identifying Steps section below for more information about supported step name and ID formats.
<PropertyLookup>	Cell C2	Indicates that this column represents the lookup string of each

File Element	Position in Example File	Purpose
column header		property. If the value is enclosed in angle brackets (< and >), the value represents an alias name. If the file was exported with type information, this column will also contain the type information after the lookup string, enclosed within curly brackets ({ and }).
<Value> column header	Cell D2	Indicates that this column represents the value of each property to be imported.
Property data for simple data types (string, number, boolean)	Row 3	Each string, numeric, or boolean property is represented by one row in the file.
Property data for containers	Rows 4, 5	Each container is represented by two rows in the file. The first row contains the names of each field in the container, and the second row contains the values of each field. Note that additional columns are used to represent container fields and values.

Identifying Steps

When the file contains data associated with step properties, either the step name, step ID, or both should be specified in the <Category> column. The following image demonstrates each method of identifying a step within a file.

	A	B	C	D
1	<PropertyGroup name='StepNameExample' id='eb6dd6d7-171c-46b4-81ce-7794aa1c0c58'>			
2	<Sequence>	<Category>	<PropertyLookup>	<Value>
3	MainSequence	My Test Step	DataSource {Expression}	Locals.TestResult
4	MainSequence	ID#:YeFRBFi25hGfSAAMKQt5BD	DataSource {Expression}	Locals.TestResult
5	MainSequence	My Test Step (ID#:YeFRBFi25hGfSAAMKQt5BD)	DataSource {Expression}	Locals.TestResult

Special Property Formats

The following sections describe the file format for properties in arrays and nested containers.

Arrays

See the following image for an example of a file containing an array of numbers:

	A	B	C	D
1	<PropertyGroup name='ArrayExample' id='eb6dd6d7-171c-46b4-81ce-7794aa1c0c58'>			
2	<Sequence>	<Category>	<PropertyLookup>	<Value>
3	MainSequence	{Locals}	SampleArray {Number [0] .. [4]}	
4	MainSequence	{Locals}	SampleArray[0]	5
5	MainSequence	{Locals}	SampleArray[1]	10
6	MainSequence	{Locals}	SampleArray[2]	15
7	MainSequence	{Locals}	SampleArray[3]	20
8	MainSequence	{Locals}	SampleArray[4]	25
9	</PropertyGroup>			

In this example, cell C3 contains the lookup string of the array, followed by information about the dimensions of the array enclosed in curly brackets { }. Rows 4-8 represent the value of each element in the array, with the value of the <PropertyLookup> column indicating which element in the array is represented by the row of data.

Nested Containers

See the following image for an example of a file with a container property that contains another container as a field:

	A	B	C	D	E
1	<PropertyGroup name='ContainerExample' id='eb6dd6d7-171c-46b4-81ce-7794aa1c0c58'>				
2	<Sequence>	<Category>	<PropertyLookup>	<Value>	
3	MainSequence	{Locals}	SampleContainer {Container}	High {Number}	Low {Number}
4	MainSequence	{Locals}	{ContainerValues}	10	1
5	MainSequence	{Locals}	SampleContainer.SubData {Container}	Voltage {Number}	Current {Number}
6	MainSequence	{Locals}	{ContainerValues}	12	0.5
7	</PropertyGroup>				

In this example, the local variable `SampleContainer`, represented on rows 3 and 4, includes a field named `SubData`, which is also a container. Note that the field names and values for the `SubData` container are represented on rows 5 and 6 of the file, below the rows for basic data type fields within the parent container.

Property Aliases

The Property Loader allows you to optionally specify alias names for properties within a file. Aliases allow you to refer to properties within the file with meaningful names, rather than a property lookup string. When a file contains aliases, a `.pla` (Property

Loader Alias) file must be associated with the file. The .pla file contains the mapping between aliases and property lookup strings.

Using Aliases within a Limits File

When aliases are used within a file, the `<PropertyLookup>` column will be replaced with an `<AliasName>` column. See the following image for an example of a file with aliases instead of property lookup strings:

	A	B	C	D
1	<PropertyGroup name='AliasExample' id='eb6dd6d7-171c-46b4-81ce-7794aa1c0c58'>			
2	<Sequence>	<Category>	<AliasName>	<Value>
3	MainSequence	My Test Step	Test Data Source {Expression}	Locals.TestResult
4	MainSequence	{Locals}	Test Result Value {Boolean}	FALSE
5	</PropertyGroup>			

Column C in this example contains the alias name for each property defined in the file.

Property Alias Files

In order for the Property Loader to load properties with alias names into a sequence file, a .pla file must be specified when the file is imported. The .pla file is encoded in XML, as seen in the following example:

```
<?xml version="1.0" encoding="utf-8"?>

<AliasItems xmlns:ts="www.ni.com/TestStand ">

  <AliasItem AliasName="Test Result Value"
    PropertyLookup="Sequence[MainSequence].Locals.TestResult"
  />

  <AliasItem AliasName="Test Data Source"
    PropertyLookup="Sequence[MainSequence].Step[My Test
    Step].DataSource" />

</AliasItems>
```

In this example, each `<AliasItem>` tag defines an alias and its associated property lookup string.

The `.pla` file will be created automatically during the export process if aliases are defined, and can be modified outside of TestStand if the alias names or property lookup strings need to be updated.

Using the Synchronization Step Type

Refer to the following topics for information on using the Synchronization step type.

Setting Up TestStand for Accessing Synchronization Objects Remotely

To create and access a synchronization object on another computer, you must configure Microsoft Windows system security on both computers. This help file uses the following terms:

- **Remote computer**—The computer that creates the synchronization object.
- **Local computer**—The computer that accesses the synchronization object.

Setting Windows System Security on the Remote Computer

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

Setting Windows System Security on the Local Computer

You must configure DCOM settings and Windows firewall settings on the computer that accesses the synchronization object remotely.

DCOM Settings for the Remote Computer - Setting Up TestStand for Accessing Synchronization Objects Remotely

Use the following information to configure the Distributed Component Object Model (DCOM) security permissions for the `TSAutoMgr.exe` server on the computer that creates the synchronization object.

Complete the following steps to configure the DCOM security permissions.

1. Log in as a user with administrator privileges.
2. Navigate to the standard Windows Control Panel facility for administrative tools and select **Component Services** or run `dcomcnfg` from the command line to launch the Component Services window.
3. On the left pane of the Component Services window, select **Component Services»Computers»My Computer**.
4. Right-click **My Computer** and select **Properties** to launch the My Computer Properties dialog box.
5. On the Default Properties tab of the My Computer Properties dialog box, enable the **Enable Distributed COM on this computer** option.



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

6. Click the **COM Security** tab of the My Computer Properties dialog box and complete the following steps to set permissions.
 - a. Click the **Edit Limits** button in the Access Permissions section to launch the Access Permission dialog box.
 - b. Click **Add** to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the `ANONYMOUS LOGON` user.
 - c. (Windows 10/8.1/7) Click **OK** to close the Select Users or Groups dialog box.
 - d. Select the user you added and enable **Remote Access** in the Permissions section.
 - e. Click **OK** to close the Access Permission dialog box.
 - f. (Windows 10/8.1/7) Click the **Edit Limits** button in the Launch and Activation Permissions section to launch the Launch and Activation Permission dialog box.
 - g. Click **Add** to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the `ANONYMOUS LOGON` user.
 - h. (Windows 10/8.1/7) Click **OK** to close the Select Users or Groups dialog box.
 - i. Select the user you added and enable **Remote Launch** and **Remote Activation** in the Permissions section.
 - j. (Windows 10/8.1/7) Click **OK** to close the Launch and Activation Permission dialog box.
7. Click **OK** to close the My Computer Properties dialog box.

8. On the left pane of the Component Services window, select **My Computer»DCOM Config** to display a list of applications on the right pane.
9. Right-click **NI TestStand AutoMgr** and select **Properties** from the context menu to launch the NI TestStand AutoMgr Properties dialog box.
10. On the **General** tab of the NI TestStand AutoMgr Properties dialog box, change the Authentication Level to **None**.
11. Click the **Security** tab of the of the NI TestStand AutoMgr Properties dialog box and complete the following steps to set permissions.
 - a. (Windows 10/8.1/7) Select the **Customize** option and click the **Edit** button in the Launch and Activation Permission section to launch the Launch and Activation Permission dialog box.
 - b. Click **Add** to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the **ANONYMOUS LOGON** user.
 - c. (Windows 10/8.1/7) Click **OK** to close the Select Users or Groups dialog box.
 - d. Select the user you added and enable **Remote Launch** and **Remote Activation** in the Permissions section.
 - e. (Windows 10/8.1/7) Click **OK** to close the Launch and Activation Permission dialog box.
 - f. Select the **Customize** option and click the **Edit** button in the Access Permissions section to launch the Access Permission dialog box.
 - g. Click **Add** to add the users you want to give remote access to. If the computer is not on a domain or the user is not a domain account, add the **ANONYMOUS LOGON** user.
 - h. (Windows 10/8.1/7) Click **OK** to close the Select Users or Groups dialog box.
 - i. Select the user you added and enable **Remote Access** in the Permissions section.
 - j. Click **OK** to close the Access Permission dialog box.
12. On the **Identity** tab of the NI TestStand AutoMgr Properties dialog box, select the **This user** option and enter a user name and password or select the **The interactive user** option. Click **OK** to close the dialog box. If you select the **The interactive user** option, a user must be logged in at the physical computer console during remote sequence execution. You can automatically log on a user each time a computer reboots by setting registry keys. Refer to Microsoft support article 315231 for more information about activating automatic logon.



Note Regardless of whether you select the **This user** option or **The**

interactive user option, the user account used on the remote computer must be an administrator account.

Related information:

- [Turn on Automatic Logon in Windows](#)

DCOM Settings for the Local Computer - Setting Up TestStand for Accessing Synchronization Objects Remotely

Complete the following steps to configure Distributed Component Object Model (DCOM) access permissions on the computer that accesses the synchronization object. You must restart TestStand for these changes to take effect.

1. Log in as a user with administrator privileges.
2. Navigate to the standard Windows Control Panel facility for administrative tools and select **Component Services** or run `dcomcnfg` from the command line to launch the Component Services window.
3. On the left pane of the Component Services window, select **Component Services»Computers»My Computer**.
4. Right-click **My Computer** and select **Properties** to launch the My Computer Properties dialog box.
5. Click the **COM Security** tab of the My Computer Properties dialog box and complete the following steps to set permissions.
 - a. Click the **Edit Limits** button in the Access Permissions section to launch the Access Permissions dialog box.
 - b. Select **ANONYMOUS LOGON** in the user name list and enable **Remote Access** in the Permissions Section. If ANONYMOUS LOGON does not appear in the user name list, click the **Add** button to add it.
 - c. Click **OK** to close the Access Permissions dialog box.

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

`ole.AuthnLevel=1`

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

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

Related concepts:

- [Effectively Using LabVIEW with TestStand](#)

Windows Firewall Settings for the Remote Computer - Setting Up TestStand for Accessing Synchronization Objects Remotely

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

- Windows 10/8.1/7 Firewall Settings

Windows 10/8.1/7 Firewall Settings - Windows Firewall Settings for the Remote Computer - Setting Up TestStand for Accessing Synchronization Objects Remotely

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

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

window.

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

Related concepts:

- [Search Directories](#)

Windows Vista Firewall Settings - Windows Firewall Settings for the Remote Computer - Setting Up TestStand for Accessing Synchronization Objects Remotely

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

1. Log in as a user with administrator privileges.
2. Navigate to the standard Windows Control Panel facility for Windows firewall and select the **Change settings** option in the Windows Firewall window to launch the Windows Firewall Settings dialog box.
3. Complete the following steps to add exceptions for the `TSAutoMgr.exe` application with the firewall enabled.
 - a. Click the **Exceptions** tab.
 - b. Click the **Add program** button to launch the Add a Program dialog box.
 - c. Click **Browse** and select `<TestStand>\Bin\TSAutoMgr.exe`.
 - d. Click **OK** to close the Add a Program dialog box.
 - e. Click the **Add port** button to launch the Add a Port dialog box.
 - f. In the **Name** control, type `DCOM`.
 - g. In the **Port Number** control, type `135`.
 - h. Select the **TCP** option and click **OK** to close the Add a Port dialog box.
4. Click **OK** to close the Windows Firewall Settings dialog box.

Related concepts:

- [Search Directories](#)

Windows Firewall Settings for the Local Computer - Setting Up TestStand for Accessing Synchronization Objects Remotely

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

- Windows 10/8.1/7 Firewall Settings

Windows 10/8.1/7 Firewall Settings - Windows Firewall Settings for the Local Computer - Setting Up TestStand for Accessing Synchronization Objects Remotely

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

1. Log in as a user with administrator privileges.
2. Navigate to the standard Windows Control Panel facility for Windows firewall and click **Allow an app or a program or feature through Windows Firewall** on the left panel of the Windows Firewall window to launch the Allowed Apps or Programs window.
3. Complete the following steps to add exceptions for the `REngine.exe` application with the firewall enabled.
 - a. Click the **Allow another app or program** button to launch the Add an App or Program dialog box.
 - b. Click **Browse** and select the client application, such as `<TestStand>\Bin\SeqEdit.exe`.
 - c. Click **Add** to close the Add an App or Program dialog box.
 - d. Click **OK** to close the Allowed Apps or Programs window.
4. Click **Advanced settings** on the left panel of the Windows Firewall window to launch the Windows Firewall with Advanced Security window.
5. Complete the following steps to specify a security exception for Distributed Connection Object Model (DCOM).
 - a. Click **Inbound Rules** on the left panel of the Windows Firewall with Advanced Security window to display the list of inbound rules.
 - b. Click **New Rule** on the Actions pane to launch the New Inbound Rule Wizard.
 - c. Select **Port** and click the **Next** button.
 - d. Select **TCP**, select **Specific local ports**, enter `135` in the Specific local ports control, and click **Next**.

- e. Select **Allow the connection** and click **Next**.
- f. Ensure that the Domain, Private, and Public options have checkmarks and click **Next**.
- g. In the Name control, enter `DCOM` and click **Finish** to close the New Inbound Rule Wizard.

Related concepts:

- [Search Directories](#)

Windows Vista Firewall Settings - Windows Firewall Settings for the Local Computer - Setting Up TestStand for Accessing Synchronization Objects Remotely

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

1. Log in as a user with administrator privileges.
2. Navigate to the standard Windows Control Panel facility for Windows firewall and select the **Change settings** option in the Windows Firewall window to launch the Windows Firewall Settings dialog box.
3. Complete the following steps to add exceptions for the client application with the firewall enabled.
 - a. Click the **Exceptions** tab.
 - b. Click the **Add program** button to launch the Add a Program dialog box.
 - c. Click **Browse** and select the client application, such as `<TestStand>\Bin\TSAutoMgr.exe`.
 - d. Click **OK** to close the Add a Program dialog box.
 - e. Click the **Add port** button to launch the Add a Port dialog box.
 - f. In the **Name** control, type `DCOM`.
 - g. In the **Port Number** control, type `135`.
 - h. Select the **TCP** option and click **OK** to close the Add a Port dialog box.
4. Click **OK** to close the Windows Firewall Settings dialog box.

Related concepts:

- [Search Directories](#)

Synchronized Sections

Place a Batch Synchronization step at the beginning of a section of steps in a sequence and specify an Enter operation for the step. Place another Batch Synchronization step at the end of the section of steps and specify an Exit operation for the step. You must place the Enter and Exit steps in the same sequence file, but you do not have to place the Enter and Exit steps in the same step group.

Each thread in a batch that enters a synchronized section blocks at the Enter step until all the other threads in the batch arrive at their respective instances of the Enter step. A thread cannot re-enter a section it has already entered. Each thread in a batch that reaches the end of the synchronized section blocks at the Exit step until all the other threads in the batch arrive at their respective instances of the Exit step.

You can use the following types of synchronized sections in sequence files:

- **Serial**—Use a Serial section to ensure that each thread in the batch executes the steps in the section sequentially and in the order you specify when you create the batch. When all threads in a batch arrive at their respective instances of an Enter step for a Serial section, TestStand releases one thread at a time in ascending order according to the order numbers you assign to the threads when you use a Batch Specification step to add the threads to the batch. As each thread reaches the Exit step for the section, the next thread in the batch proceeds from the Enter step. After all the threads in the batch arrive at the Exit step, the threads exit the section together.
- **Parallel**—Use a Parallel section to run each thread independently. When all threads in a batch arrive at their respective instances of an Enter step for a Parallel section, TestStand releases all the threads at once. As each thread reaches the Exit step for the section, the thread blocks until all the threads in the batch reach the Exit step. After all the threads in the batch arrive at the Exit step, the threads exit the section together.
- **One Thread Only**—Use a One Thread Only section to specify that only one thread in the batch executes the steps in the section. Typically, you use this type of section to perform an operation that applies to the batch as a whole, such as

raising the temperature in a test chamber. When all threads in a batch arrive at their respective instances of an Enter step for a One Thread Only section, TestStand releases only one thread. When that thread arrives at the Exit step for the section, all remaining threads in the batch jump from the Enter step to the Exit step, skipping the steps within the section. All the threads in the batch exit the section together.



Note For nested sections, TestStand ignores the inner section when the type of the outer section is set to `One Thread Only`.

Related information:

- [Batch Specification Step](#)
- [Batch Synchronization Step](#)
- [Semaphore Step](#)

Mismatched Sections - Synchronized Sections

Sections become mismatched when all the threads in a batch block at different Enter or Exit operations. This situation can occur when a sequence implements a conditional flow of execution as a result of preconditions, post actions, or other flow control operations.

When TestStand detects mismatched sections, the thread at the Enter or Exit step that appears earliest in the hierarchy of sequences and subsequences proceeds as if all threads in the batch are at the same step.

When multiple Enter and Exit operations are equally early in the hierarchy of sequences and subsequences, Enter operations proceed first.

Nested Sections - Synchronized Sections

Nested sections can occur within the same sequence or when you call a subsequence inside a synchronized section and the subsequence also contains a synchronized section. You must exit nested sections in the reverse order in which you entered the sections.

When you nest one section inside another, TestStand honors the inner section only when the type of the outer section is serial or parallel. TestStand ignores the inner section when the type of the outer section is One Thread Only. For example, when you nest one serial section in another serial section, each thread that enters the outer serial section proceeds only until the Enter step of the inner section and then waits for the other threads to reach the same Enter step of the inner section before completing the inner section.

Using the Data Streams Step Type

Refer to the following topics for information on using the Data Streams step type.

Attaching a CSV Output Stream to the Current Execution

When you create a CSV file with the New CSV Output Stream step type, you can attach it the current execution if you name the stream. The name must be unique within the scope of the execution. Providing this name (or the optional `ExecutionOutputRecordStream` object reference) to a Write Record step allows you to display the data being written to the CSV file in the Report view in real time.

You can limit the amount of data displayed in the Report view with the **Records to Display** option. Specify a positive value to display only the most recently written records. Specify -1 to display all records. Specify 0 to disable updates to the Report view which will improve performance.



Note Specifying -1 means that the memory used by the Report view grows linearly with the amount of data written.

Although you can view data written to an output CSV file in the Report view, the CSV file is not a TestStand report as it is not otherwise connected to TestStand reporting features. TestStand reports are driven by process models with their data coming from step results. Using Data Streams, sequences can create and store data to CSV files without running in a TestStand process model. If they do run in a process model, they can still generate ATML, XML, ASCII, HTML, or custom reports. Reports generated using the process model do not interfere with the CSV output file, and the CSV output file does not interfere with reports generated by the model. There is no implicit connection between step results and data written to CSV files. Only data specifically

written using Data Streams step types or the TestStand API is stored to the CSV file.

For a demonstration of TestStand Data Streams, see the <TestStand Public>\Examples\Fundamentals\Using Data Streams examples.

Related concepts:

- [Using Data Streams](#)

Reading from CSV Files with Data Streams Step Types

Use the New CSV Input Stream step type to open a CSV file for reading and store a reference to it. In addition to specifying the path to the input file, you can specify a tag to search for and/or some number of lines to skip. These options enable reading from files that are not completely homogenous. For example, it is common for a CSV file to have metadata at the top (the date, time, operator id, location, etc.) followed by column headers, and then the main data table. The combination of scanning for a specific string and/or skipping a fixed number of lines allows the New CSV Input Record Stream step type to prepare such a file for reading with a for each loop. You can use the API to read more complex files.

After you open a CSV file, you can iterate over its rows using the For Each step type. Each iteration of the loop reads one row of the CSV file and stores the data in a container or array you specify. If you specify a container, the container represents a row, and each subproperty corresponds to a column. If you specify an array, the entire array represents a row, and each array element corresponds to a column. The loop continues until the For Each step encounters a blank line or the end of file.



Note Because the For Each step reads only a single record during each iteration, it is possible to iterate over large CSV files. The loop itself requires only enough space to store a single row of data.

You must close the CSV file when you are finished. You can close the CSV file by calling the `InputRecordStream.Close()` method through the API. Alternatively, you can configure the For Each step type to close the file automatically when the loop terminates.

Writing to CSV Files with Data Streams Step Types

Use the New CSV Output Stream step type to create a CSV file for writing and store a reference to it. The file open mode controls the behavior if the specified file already exists. After you open the file, you can write to it using the Write Record step type or the TestStand API.

Each invocation of a Write Record step writes a single row of data to the output CSV file. You can specify a row using either a container or an array. If you specify a container, the subproperties of the container correspond to columns in the CSV file. If you specify an array, each array element corresponds to a column in the CSV file.

You can also write to the output file using the TestStand API. See for example `OutputStream.WriteRecord()`, `CsvFileOutputStream.WriteLine()`, and `CsvFileOutputStream.WriteFieldHeaders()`. You must close the CSV file when you are finished with it by calling the `OutputStream.Close()` method.

Module Adapters

The TestStand Engine uses module adapters to invoke code modules that sequences call. A code module is a program module from an ADE or programming language and can contain one or more functions that perform a specific test or other action. Module adapters load and call code modules, pass parameters to code modules, and return values and status from code modules. TestStand handles core test management functionality, including the definition and execution of the overall testing process, user management, report generation, database logging, and more.

TestStand supports the following module adapters:

- ActiveX/COM Adapter
- C/C++ DLL Adapter
- HTBasic Adapter
- LabVIEW Adapter
- LabWindows/CVI Adapter
- .NET Adapter
- Sequence Adapter
- Python Adapter



Note (64-bit TestStand) The HTBasic Adapter is not supported.

Select **Configure»Adapters** to configure an adapter.

To specify a code module for a step, use a Module tab of the Step Settings pane in the TestStand Sequence Editor and a Specify Module dialog box in a TestStand User Interface. The actual title of the Specify Module tab or dialog box is different for each adapter.

Adapters specific to an application development environment (ADE) can open the ADE, create source code for a new code module in the ADE, and display the source for an existing code module in the ADE. The adapters support stepping into the source code in the ADE while you execute the step from the TestStand Sequence Editor or a TestStand User Interface.

Source Code Templates

For each adapter that supports source code templates, the Module tab and the Specify Module dialog box display a command button for creating source code based on a template for the step. If more than one template is available for the adapter and step type combination you use to create the step, you can select from a list of the templates. If only one template is available, the adapter uses the template without giving you additional options.

Related concepts:

- [Code Modules](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Programming with the TestStand API in LabWindows/CVI](#)
- [Adapter and Code Module Support for 64-bit TestStand](#)
- [Driver Support for 64-bit TestStand](#)

Search Directories

TestStand includes a list of search directories module adapters use to resolve relative paths of code modules for steps and substeps in step types and to locate code

modules when executing steps. TestStand also uses the search directories to resolve relative pathnames for files and directories when calling the TestStand API `Engine.FindFileEx` and `Engine.FindPath` methods.

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

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

- Current sequence file directory
- Current workspace directory
- Application directory (disabled by default)
- `<TestStand Public>` directory
- `<TestStand>` directory
- `<TestStand>\Bin` directory
- Initial working directory (disabled by default)
- Windows system directory
- Windows directory
- PATH environment variable (disabled by default)
- `<TestStand Public>\Components` directory
- `<TestStand>\Components` directory

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

If the search directory is non-recursive, TestStand appends the relative path to the search directory path to determine if the file exists. As soon as TestStand finds the file, the search stops.

If the search directory is recursive, TestStand appends the relative path to all the

subdirectories too. NI recommends using the recursive option sparingly to avoid performance issues and to reduce the risk of matching incorrect files with the same name.

Related concepts:

- [Using Substeps](#)
- [TestStand Directory Structure](#)
- [Configuring Search Directories for Deployment](#)
- [Locating the Correct Code Module in 32-bit TestStand and 64-bit TestStand](#)
- [Using TestStand Search Directories to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand](#)

Configuring Search Directories for Deployment

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

NI recommends installing the <TestStand Application Data>\Cfg\SearchDirectories.cfg file with the deployment for that system because TestStand stores the search directories in SearchDirectories.cfg. Use the **Force File to Install** option on the Distributed Files tab of the TestStand Deployment Utility to overwrite existing files.

Related concepts:

- [Preparing Source Components for Deployment](#)
- [TestStand Directory Structure](#)
- [Deploying TestStand Systems](#)

Calling DLLs from TestStand

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

Calling LabVIEW DLLs that Use ActiveX Controls

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

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

Calling DLLs that Use MFC Functions

The MFC Library places several requirements on DLLs that use the DLL version of the MFC run-time library. When you call a DLL that includes MFC functions, verify that the DLL meets these requirements. Also, when the DLL uses resources such as dialog boxes, verify that the `AFX_MANAGE_STATE` macro appears at the beginning of the function body of each function you call. Refer to the MFC documentation for more information about calling DLLs.

Related concepts:

- [Loading Subordinate DLLs](#)
- [Launching an MFC Dialog with ActiveX Controls](#)

Loading Subordinate DLLs

TestStand directly loads and runs the DLLs you specify on the C/C++ DLL Module tab for the C/C++ DLL Adapter or the LabWindows/CVI Module tab for the LabWindows/CVI Adapter. Because code modules most likely call subsidiary DLLs, such as instrument drivers, you must ensure that the operating system can find and load any DLL you specify.

The adapter attempts to load subordinate DLLs using the following search directory precedence:

1. The directory that contains the DLL the adapter calls directly
2. The `Windows\System32` and `Windows\System` directories
3. The `Windows` directory
4. The current working directory of the application
5. The directories listed in the PATH environment variable

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

1. The directory that contains the application that loaded the adapter
2. The `Windows\System32` and `Windows\System` directories
3. The `Windows` directory
4. The current working directory of the application, which the adapter sets to the directory that contains the DLL it calls directly
5. The directories listed in the PATH environment variable



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

Use the TestStand Deployment Utility to deploy code modules and subsidiary DLLs for use with TestStand.

Related concepts:

- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)
- [Programming with the TestStand API in LabWindows/CVI](#)
- [Code Modules](#)
- [Deploying TestStand Systems](#)

Choosing Between the Shared or Side-by-Side Version of the LabWindows/CVI Run-Time Engine

The shared and side-by-side versions of the LabWindows/CVI Run-Time Engine (RTE) differ in the following ways when you install a new version of the RTE.

- **Shared RTE**—Installing a new version of the default shared RTE upgrades the previous version of the RTE. The default RTE is backward compatible and can execute LabWindows/CVI executables or DLLs built with previous versions of LabWindows/CVI. However, the executables or DLLs might not behave exactly the same way as a result of changes in the RTE code. For example, optimizations in the RTE might result in functions executing faster than in previous versions of the RTE.
- **Side-by-Side RTE**—Because the side-by-side version of the RTE is version-specific, installing a new version of the side-by-side RTE prevents any behavior changes to applications built with older versions of LabWindows/CVI. Previously built LabWindows/CVI applications behave identically because they continue to call the same version of the side-by-side RTE they originally called.

For most applications, NI recommends that you use the default shared RTE. However, if the test system requires validation, use the side-by-side version of the RTE to minimize the amount of validation required. For example, if you start using a new LabWindows/CVI feature in a code module, you must install a new version of the LabWindows/CVI RTE on test stations that use the test system. If you use the shared RTE, you would validate the entire test system after you installed a new LabWindows/CVI RTE because the behavior of DLLs could change using the new version of the RTE. If you use the side-by-side RTE, you would validate only the new code because the old code continues to call the same RTE version it did before.

Related concepts:

- [Binding a TestStand Custom User Interface to a Side-by-Side Version of the LabWindows/CVI RTE](#)
- [Identifying Components to Deploy](#)

Using Side-by-Side Versions of the LabWindows/CVI Run-Time Engine with TestStand

Typically, the default LabWindows/CVI Run-Time Engine (RTE) is appropriate for most DLLs and applications you build in LabWindows/CVI to use with TestStand. However, on occasion, you might want to isolate a test system from upgrades or downgrades to the LabWindows/CVI RTE. With LabWindows/CVI 2012 or later you can isolate LabWindows/CVI applications and DLLs from upgrades or downgrades to the LabWindows/CVI RTE by binding an executable or DLL to a specific, side-by-side

version of the LabWindows/CVI RTE. Refer to the Binding Executables and DLLs to Side-by-Side Run-Time Engines topic in the LabWindows/CVI Help for more information about binding executables and DLLs to a side-by-side version of the LabWindows/CVI RTE.

Calling DLLs that Use a Side-by-Side Version of the LabWindows/CVI RTE in TestStand

You can use the LabWindows/CVI Adapter and the C/C++ DLL Adapter in TestStand to load and call DLL code modules that use a side-by-side LabWindows/CVI RTE or the default LabWindows/CVI RTE within the same sequence file. However, TestStand does not support calling DLLs you create with tools other than LabWindows/CVI, such as Microsoft Visual Studio, to use a side-by-side LabWindows/CVI RTE.

Related concepts:

- [Programming with the TestStand API in LabWindows/CVI](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)
- [Binding a TestStand Custom User Interface to a Side-by-Side Version of the LabWindows/CVI RTE](#)
- [Choosing Between the Shared or Side-by-Side Version of the LabWindows/CVI Run-Time Engine](#)
- [Identifying Components to Deploy](#)

Reading Parameter Information for a DLL

When a DLL contains export information or when a DLL file contains a type library, the LabWindows/CVI and C/C++ DLL Adapters automatically populate the Function control on the Module tab of the step with all the function names exported from the DLL.

In addition, when you select a function in the DLL, the adapter queries the export information or the type library for the parameter list information and displays it in the Parameters Table control on the Module tab. When the adapter cannot determine parameter information, you must enter the parameter information manually.

Related concepts:

- [Programming with the TestStand API in LabWindows/CVI](#)

- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)
- [Adding Type Libraries to LabWindows/CVI DLLs](#)
- [Generating Type Library Information to Include in a DLL](#)

Debugging DLLs

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

Debugging LabVIEW DLLs

To build a debuggable shared library in the LabVIEW development system, you must configure the shared library to enable debugging in the build specification for the shared library before building the shared library. Using the LabVIEW development system, you can then connect to the TestStand application process that loads and calls the VIs in the debuggable shared library.

Debugging LabWindows/CVI DLLs

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

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

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

The LabWindows/CVI debugger automatically attaches to a TestStand process when you perform a step into operation on a LabWindows/CVI Adapter step. If the LabWindows/CVIDLL is debuggable and the source code is available, LabWindows/CVI

suspends the processing the called DLL function; otherwise, TestStand performs a step over operation. You do not need to launch the Sequence Editor from LabWindows/CVI to access the debugger or choose to attach the debugger to an existing process.



Note You must be using LabWindows/CVI 2015 SP1 or later for the debugger to automatically attach to a TestStand process.

Debugging Microsoft Visual Studio DLLs

To step into a DLL directly from TestStand into a supported version of Visual Studio, you must configure the step to use the C/C++ DLL Adapter. When you attempt to step into a DLL while Visual Studio is not attached to the TestStand process, TestStand launches Visual Studio, which automatically attaches to the TestStand process using native debugging.

When you launch Visual Studio from TestStand, Visual Studio runs with the same privileges you used to run TestStand. If you launch TestStand while logged in as a user with standard privileges and then you launch Visual Studio from TestStand, you cannot execute tasks in Visual Studio that require administrator privileges. Refer to the LabWindows/CVI and Visual Studio documentation for more information about debugging DLLs in an external process.



Note

- Save sequence files and workspaces before you stop debugging and terminate the TestStand process because most ADEs terminate the process without prompting you to save modified files in TestStand.
- When you use LabWindows/CVI to debug a DLL in the TestStand process, you cannot debug a .NET assembly at the same time. When you use Visual Studio to debug an assembly in TestStand and you want to use LabWindows/CVI to debug code modules at the same time, you must configure the LabWindows/CVI Adapter to execute the steps in an external instance of LabWindows/CVI.
- You must attach Visual Studio to the TestStand process correctly depending on the type of code module you want to debug. For C++

code modules, NI recommends using only native debugging when attaching Visual Studio.

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

Related concepts:

- [Programming with the TestStand API in LabWindows/CVI](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)
- [Getting Started with TestStand](#)

Adding Type Libraries to LabWindows/CVI DLLs

When a DLL contains export information or when a LabWindows/CVI DLL file contains a type library, the LabWindows/CVI Adapter automatically populates the Function control on the LabWindows/CVI Module tab with all the function names the DLL exports.

In addition, when you select a function in the DLL, the adapter queries the export information or the type library for the parameter list information and adds it in the Parameters Table control on the LabWindows/CVI Module tab.

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

Related concepts:

- [Programming with the TestStand API in LabWindows/CVI](#)
- [Generating Type Library Information to Include in a DLL](#)
- [Specifying String Buffer Size in a Type Library](#)

Generating Type Library Information to Include in a DLL

LabWindows/CVI can use the information specified in a function panel file to generate type library information to include in a DLL. Complete the following steps to generate a

type library resource from a function panel and add the type library resource to a DLL.

1. Open a new function panel file and create a function panel for each exported function you want to include in the type library.
2. Add the function panel file to the LabWindows/CVI project.
3. In the LabWindows/CVI project window, select **Build»Target Settings** to launch the Target Settings dialog box.
4. In the Target Settings dialog box, click the **Type Library** button to launch the Type Library dialog box.
5. In the Type Library dialog box, enable the **Add type library resource to DLL** option and enter the path to the file in the Function panel file control.
6. Click **OK** to close the Type Library dialog box and to close the Target Settings dialog box.
7. In the Project window, select **Build»Create Debuggable Dynamic Link Library** to build the DLL. Click **OK** when LabWindows/CVI prompts you that the files are created.

LabWindows/CVI imposes certain requirements on the declaration of the DLL API in a type library. Use the following guidelines to ensure that TestStand can use the DLL:

- Use typedefs for structure, union, and enumeration parameters.
- Do not use structures that require forward references or that contain pointers.
- Do not use pointer types except when passing parameters by reference.

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

Related concepts:

- [Adding Type Libraries to LabWindows/CVI DLLs](#)
- [Specifying String Buffer Size in a Type Library](#)

Specifying String Buffer Size in a Type Library

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

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

Related concepts:

- [Adding Type Libraries to LabWindows/CVI DLLs](#)
- [Generating Type Library Information to Include in a DLL](#)

Module Adapter Parameter Mapping Guidelines

When you load a new prototype for any adapter, each adapter uses the following guidelines to copy the configured argument values for parameters from the old prototype to the new prototype:

Adapter	Rules (in Order of Priority) for Mapping Arguments
LabVIEW	1. Map arguments based on parameter name and compatible type. 2. Map arguments based on connector pane position and compatible type. 3. Map arguments based on compatible type, same input or output direction, and matching the prefix of the old parameter name with the new parameter name. 4. Map arguments based on compatible type, same input or output direction, and matching the prefix of the new parameter name with the old parameter name. 5. Use the map the step type specifies, if available.

Adapter	Rules (in Order of Priority) for Mapping Arguments
LabWindows/ CVI	1. Map arguments based on parameter name and compatible type. 2. Map arguments based on position and compatible type.
C/C++ DLL	Does not specify any parameters or mapping rules.
.NET	
ActiveX/COM	
Sequence	
HTBasic	
Python Adapter	Map arguments based on parameter name and position.

Related concepts:

- [Module Adapters](#)

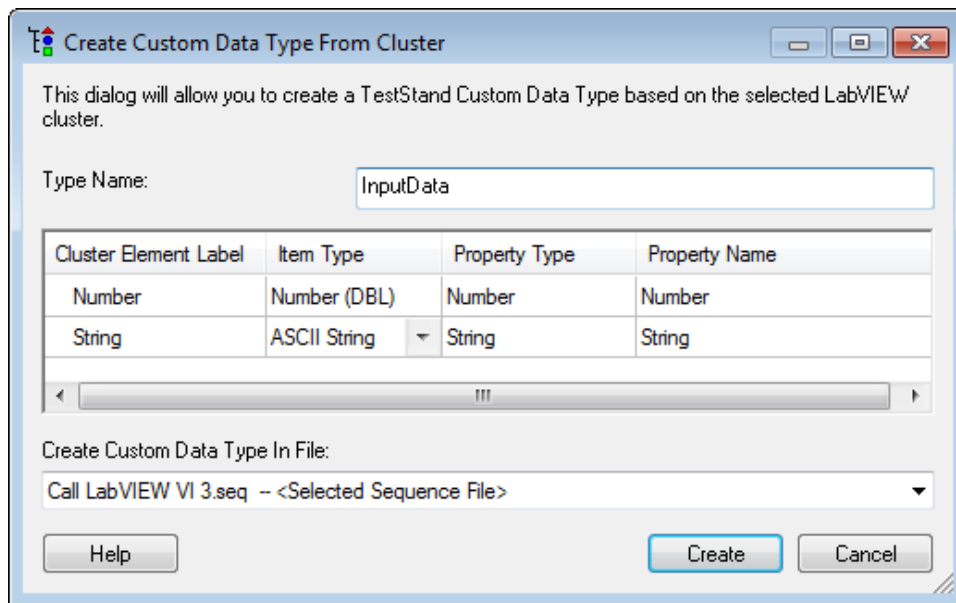
Passing Existing TestStand Container Variables to LabVIEW

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

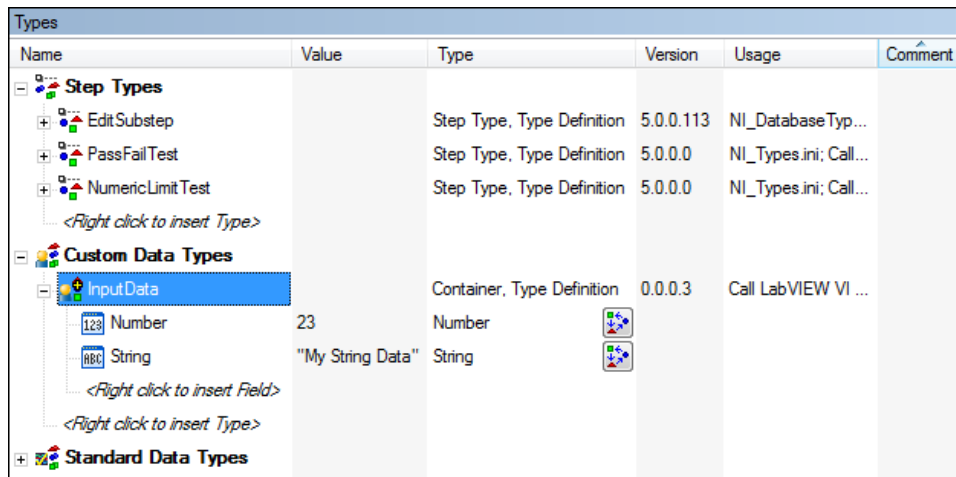
1. Click the **Create/Update Custom Data Type**  button, located in the **Type** column of the **VI Parameter Table** on the LabVIEW Module tab, to launch the Create/Update Custom Data Type from Cluster dialog box.
2. create a TestStand custom data type, such as a container, that matches an existing LabVIEW cluster.
3. Use the LabVIEW Cluster Passing tab on the Type Properties dialog box for the new custom data type to specify how TestStand maps subproperties to elements in a LabVIEW cluster.

This example illustrates how to create an InputData custom data type to map cluster data to TestStand, and create a ContainerData local variable that is an instance of the InputData custom data type.

The following figure shows the Create/Update Custom Data Type from Cluster dialog box for the InputData data type:



The following figure shows the custom InputData data type on the Types window:



The following figure shows a **VI Parameter Table** in which the **Input Cluster** parameter data passes to the ContainerData local variable:

Parameter Name	Type	In/Out	Log	Default	Value	
Input Cluster	Container	in	☐	☐	Locals.ContainerData	☐ ☒
Number	Number (DBL)	in	☐	☐	Locals.ContainerData.Number	☐ ☒
String	ASCII String	in	☐	☐	Locals.ContainerData.String	☐ ☒
PASS/FAIL Flag	Boolean	out	☐		Step.Result.PassFail	☐ ☒
Report Text	ASCII String	out	☐		Step.Result.Report Text	☐ ☒
error out	Container	out	☐		Step.Result.Error	☐ ☒

When you specify the data to pass for a cluster parameter, use a variable that is an instance of the new custom data type.



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

Related concepts:

- [Passing Structs to Code Modules - LabVIEW DLL](#)

Updating Cluster Element Mapping

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

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

LabVIEW Adapter

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

- Call VIs with arbitrary connector panes.
- Call VIs that exist on disk, in LLBs, or in LabVIEW packed project libraries. You can also call VIs in the context of a LabVIEW project or call LabVIEW class member VIs.
- Call member VIs from LabVIEW classes using LabVIEW dynamic dispatching, when required.
- Call VIs on remote computers.
- Configure and call LabVIEW Property Nodes using I/O References or LabVIEW classes.



Note You must have LabVIEW 2013 or later to configure and call LabVIEW Property Nodes in TestStand.

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



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

When you specify a module for a step, the TestStand Sequence Editor displays the LabVIEW Module tab and TestStand User Interfaces launch the Edit LabVIEW VI Call dialog box.

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

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

Related concepts:

- [Adapter and Code Module Support for 64-bit TestStand](#)
- [Calling LabVIEW Class Member VIs from TestStand](#)
- [Calling VIs on Remote Computers](#)
- [Calling LabVIEW VIs](#)
- [Cross-Bitness Support using the LabVIEW Development System](#)
- [Driver Support for 64-bit TestStand](#)
- [Express VIs and Property Node Calls in 32-bit TestStand and 64-bit TestStand](#)
- [LabVIEW Code Module Support for 64-bit TestStand](#)
- [Module Adapter Parameter Mapping Guidelines](#)
- [Migrating LabVIEW Code Modules from 32-bit TestStand to 64-bit TestStand](#)
- [Organizing LabVIEW-Based TestStand Systems](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Simultaneously Supporting 32-bit TestStand and 64-bit TestStand](#)

Related information:

- [LabVIEW User Manual](#)

Caveats for Configuring the LabVIEW Adapter to Use a LabVIEW Run-Time Engine or Other Executable Server

When you select LabVIEW Run-Time Engine or Other Executable as a server in the LabVIEW Adapter Configuration dialog box, you must ensure that the server can locate the complete hierarchy of VIs, including any subVIs, that TestStand executes. The version of the VIs that TestStand executes must match the version of the LabVIEW RTE or the version of LabVIEW that built the executable.



Note TestStand can use newer versions of the LabVIEW RTE when you install LabVIEW on your development system. You can include newer versions of the LabVIEW RTE in deployments using the Drivers and Components dialog box of the TestStand Deployment Utility. If you do not have a LabVIEW development system installed on the computer, the default value of the LabVIEW RTE option is **Auto detect using VI version**, and you do not need to deploy the `Adapters.cfg` file that specifies the LabVIEW RTE version to use.

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

This behavior can also occur in the following situations:

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

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

Normally, the LabVIEW RTE can execute top-level VIs only when you save the VI with the entire VI hierarchy or when the LabVIEW RTE can locate the subVIs using the LabVIEW search directories. By default, the LabVIEW RTE does not search for required subVIs within the LabVIEW development system, such as files in `vi.lib`. However, when TestStand loads a LabVIEW RTE on a computer where you also installed the LabVIEW development system, TestStand automatically adds directories from the development system to the search paths for the LabVIEW RTE, but only for directories where the version of the development system matches the version of the LabVIEW RTE. Therefore, when TestStand executes a top-level VI saved without a hierarchy, the LabVIEW RTE searches the directory of the top-level VI and then searches the directories TestStand added to find subVIs. When the LabVIEW RTE finds a subVI saved in a different version of LabVIEW first, the LabVIEW RTE cannot load the subVI and does not load the top-level VI.

Related concepts:

- [Search Directories](#)

Related information:

- [LabVIEW Adapter Configuration Dialog Box](#)

Normalization of Numeric Values Passed to Enumeration Parameter Values

When you use TestStand to specify a VI for a LabVIEW step code module and the VI contains enumeration parameters but you pass a numeric value that is outside the range of valid enumeration values to an enumeration parameter, LabVIEW modifies the value to be within the valid range before executing the VI, as the following table describes:

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

Related concepts:

- [Code Modules](#)

Enumeration Parameter Value Update Behavior

If you change the number of items in an enumeration in LabVIEW, you must reload the VI prototype for a VI that uses the enumerated values as parameters in TestStand before you can execute the sequence.

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

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

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

When the Enumeration Is an Instance of a LabVIEW Type Definition

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

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

Change You Make to LabVIEW Type Definition	Effect You See in TestStand VI Parameter Table	Effect You See in TestStand Step Properties
different position in the list of enumeration values.	label.	label and updates the corresponding numeric value.
You delete the string label and the numeric value.	TestStand updates the string label to use the last enumeration value in the list.	TestStand updates the numeric value and updates the string label to use the last enumeration value in the list.

When the Enumeration Is Not an Instance of a LabVIEW Type Definition

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

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

LabVIEW Data Types in TestStand

TestStand provides number, string, Boolean, and object reference built-in data types. TestStand also provides several standard named data types, including Path, Error, LabVIEWAnalogWaveform, and others. You can create container data types to hold any number of other data types. TestStand container data types are analogous to LabVIEW clusters. You can use references to external objects, such as ActiveX (Microsoft ActiveX) objects or VISA sessions, between different types of code modules.

LabVIEW includes a greater variety of built-in data types than TestStand does, so TestStand converts LabVIEW data types in certain ways when calling VIs.

Converting LabVIEW Numeric Representations

The following table shows how TestStand converts LabVIEW numeric representations:

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

Converting LabVIEW Controls and Indicators

The following table shows how TestStand converts LabVIEW controls or indicators:

LabVIEW Control or Indicator	TestStand Data Type
Enum (U32, U16, or U8)	Enumeration
Ring control (Text Ring, Menu Ring, Text and Picture Ring, or Picture Ring)	Enumeration
String	String
Path	Path or string
ActiveX Control or Automation Refnum	Object reference
.NET Refnum	Object reference You can pass .NET references created in LabVIEW to TestStand and store the references in Object Reference variables in TestStand and then pass them to other VIs. You can also share objects that derive from MarshalByRefObject or that are serializable between LabVIEW and TestStand. In TestStand, you can use those references with the .NET Adapter. Similarly, in LabVIEW, you can directly use references you create with the TestStand .NET Adapter. Refer to Microsoft MSDN documentation for more information about MarshalByRefObject and serializable objects.  Note Passing data between LabVIEW and TestStand is not supported in TestStand 2024 Q4 and subsequent versions.
Waveform	LabVIEWAnalogWaveform To create variables for I64 or UI64 Waveforms, you must create a new custom data type, because the representation of the Y element that corresponds to the standard LabVIEWAnalogWaveform data type is set to double-precision 64-bit floating-point.
Digital waveform	LabVIEWDigitalWaveform

LabVIEW Control or Indicator	TestStand Data Type
Digital data	LabVIEWDigitalData
Picture	String You must select Binary String from the ring control in the Type column of the VI Parameter Table on the LabVIEW Module tab.
Refnum (File I/O, VI, Menu, Queue, TCP connection, and so on)	Number You cannot use references to internal LabVIEW objects inside TestStand or in other types of code modules. You can store only references to LabVIEW objects in TestStand properties and then pass the properties to other VIs.
Timestamp	String TestStand converts the LabVIEW Timestamp data type to a System Time data type and then to a string data type with the <code>m/d/yyyy h:mm:ss.sss AM/PM</code> format, where TestStand rounds fractions of a second to the nearest millisecond.
m	Month as number without leading zero
d	Day of the month as number without leading zero
YYYY	Four-digit year
h	Hour of the day, without leading zero (12-hour clock)
mm	Minute of the hour, with leading zero
ss	Second of the minute, with leading zero
sss	Milliseconds of the second, with leading zero
AM/PM	AM or PM uppercase
Error I/O	Error By default, when a VI uses the standard LabVIEW error out cluster as an output

LabVIEW Control or Indicator	TestStand Data Type
	parameter, TestStand automatically passes that value into the <code>Step.Result.Error</code> property for the step. You can also update the value manually. If an error occurs during execution of the VI and the error out cluster is passed to <code>Step.Result.Error</code> , TestStand launches the Run-Time Error dialog box by default.
Array of x	Array of TestStand (x)
Variant	Any TestStand data type  Note Passing LabVIEW I/O references (DAQmx Task Name, DAQmx Channel Name, VISA Resource Name, IVI Logical Name, FieldPoint IO Point, or Motion Resource) through TestStand as a variant is not supported. Use the LabVIEWIOControl instead.
LabVIEW Object	Object reference
Cluster	Container
I/O controls (DAQmx Task Name, DAQmx Channel Name, VISA Resource Name, IVI Logical Name, FieldPoint IO Point, or Motion Resource)	LabVIEWIOControl
IMAQ Session	Number
Other I/O controls (DAQmx Physical Channel Name, Terminal Name, Analog Trigger Source, Scale Name, Device Name, or Switch Name)	String



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

Related concepts:

- [Code Modules](#)
- [Passing an Instance of a LabVIEW Class Object](#)

Related information:

- [PropertyObject](#)

Legacy Configurations for LabVIEW Code Modules in TestStand

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

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

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

Although you usually create new VIs using the LabVIEW Module tab for steps that use

the LabVIEW Adapter, you can configure the LabVIEW Adapter to create legacy-style VIs.

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

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



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

Related concepts:

- [TestStand ActiveX API Overview](#)

Related information:

- [LabVIEW Adapter Configuration Dialog Box](#)
- [Legacy VI Settings Dialog Box](#)

Error Reporting for Broken VIs in the LabVIEW Run-Time Engine

The LabVIEW Adapter provides detailed information about broken VIs when used with the LabVIEW 2018 or later run-time engine. The error information is displayed in the error tooltip in the Step Settings Pane and when you execute the step.

Detailed information about the error is also logged into a text file, named the same as the VI name configured in the TestStand step. The log file is saved to the TestStand temp directory (<TestStand_Temp>), which is in the TestStand subdirectory in the directory the TEMP environment variable specifies, which by default is %USERPROFILE%\AppData\Local\Temp.

Version Independent Run-Time Engine Support

Version Independent Runtime Engine support was introduced in LabVIEW 2017 and allows the LabVIEW runtime to load source files, such as VIs and projects, and build outputs, such as packed project libraries (PPLs), DLLs, and executables that were saved or built in an older version of LabVIEW.

For example, a PPL built in LabVIEW 2017 can now be loaded and executed by the LabVIEW 2018 runtime.

TestStand 2016 SP1 supports this feature in the LabVIEW Adapter. Enabling this feature in TestStand reduces the number of LabVIEW runtime engines that would have to be loaded into the TestStand process to execute LabVIEW code modules.

PPL build specifications in LabVIEW also have an option to specify whether the build output subscribes to this feature. LabVIEW enables these options by default for PPL build specifications you create in LabVIEW 2017 and later.

Disabling the option binds a PPL build specification to a specific version of LabVIEW, prevents any changes to the performance profiles, and helps you avoid unexpected problems resulting from compiler upgrades.

For more information about this feature in LabVIEW, refer to the LabVIEW Help.

TestStand allows users to enable or disable this feature in LabVIEW using the **Enable Version Independent Runtime Engine** LabVIEW Adapter option.



Note Changes to this option do not take effect until you restart the TestStand application.

When you enable this adapter option, Version Independent Runtime Engine support is

enabled for all instances of LabVIEW runtimes, version 2017 or later, that are used by the TestStand application. TestStand can load and execute LabVIEW files saved in LabVIEW 2017 or later in a later version of the LabVIEW runtime.

For example, if a TestStand sequence calls LabVIEW code modules saved in LabVIEW 2017 and Version Independent Runtime Engine support is enabled, then these LabVIEW files will be loaded and executed in the LabVIEW 2018 Runtime Engine.

When Version Independent Runtime Engine support is disabled, the version of the VI called in TestStand, and all its dependencies, must match the version of the runtime being used to load or run the VI.

When Version Independent Runtime Engine support is enabled, and you are using LabVIEW 2018 or later runtimes and loading files saved in LabVIEW 2017 or later, this restriction no longer applies.



Note

- The LabVIEW 2017 runtime does not allow the loading of VIs, PPLs, or DLLs saved in LabVIEW 2016 or earlier, even if Version Independent Runtime Engine support is enabled. The support for this feature in TestStand applies only to LabVIEW 2018 and later runtimes when loading LabVIEW files saved in LabVIEW 2017 or later.
- The Version Independent Runtime feature is enabled in LabVIEW Development System, version 2017 and later, by default. TestStand 2019 and later versions will not modify this setting on the active LabVIEW Development System.

Errors on Loading a VI with Version Independent Runtime Engine Support Enabled

If a VI is reported as broken in TestStand when Version Independent Runtime Engine support is enabled, complete the following steps to resolve the issue:

1. Open the VI in the LabVIEW development system. If the VI is broken, fix any errors in the VI and save it.
2. Ensure that neither the VI nor any of its dependencies are saved in LabVIEW 2016

or earlier.

3. If the VI is not broken in the LabVIEW development system when Version Independent Runtime Engine support is disabled, the issue may be that the VI has dependencies in LabVIEW folders such as `vi.lib`, `user.lib`, or `instr.lib`. To resolve this issue, NI recommends that you create a deployment of your sequence files and VIs and execute the sequence files from the deployed image. Use the TestStand Deployment Utility to create a deployment of your sequence file(s) and LabVIEW VIs.

Related concepts:

- [TestStand Deployment Utility](#)

Related information:

- [LabVIEW Adapter Configuration Dialog Box](#)

Switching between VIs and Packed Project Libraries

The LabVIEW Adapter allows you to easily switch the configuration of a step between executing a VI and executing a VI in a packed project library without having to modify the step. You can also configure the LabVIEW Adapter to automatically build a missing or out-of-date packed project library at the start of execution.

In previous versions of TestStand, after you build the packed project library in LabVIEW, you would have to modify the step to point to the VI in the packed project library. Now, if the step is configured to call a VI in the context of a project and the project has a build specification configured to build a packed project library containing that VI, you can choose to execute the VI in the packed project library instead of the VI.

1. Set the following settings in the Override Module Settings window for a step calling a VI:
 - a. Select the build specification for the VI.
 - b. Set the path of the VI in the packed project library.
 - c. Optional: Set the path of the project to use when executing the VI in the packed project library.
2. Enable the **Always run VI in Packed Project Library** option in the LabVIEW Adapter Configuration dialog box or step settings.



Note Enabling **Always run VI in Packed Project Library** in the LabVIEW Adapter Configuration dialog box overrides the module settings for all LabVIEW steps and always run the VI in the packed project library. You must configure the VI settings in the Override Module Settings window to use this option.



Note

The LabVIEW Adapter will automatically run the VI in the packed project library and project configured in the Override Module Settings window instead of the VI and project configured in the step.

3. If LabVIEW Development System is installed, you can enable automatically building the packed project library at the start of execution by enabling the **Automatically Build Packed Project Libraries at the Start of Execution** option in the LabVIEW Adapter Configuration dialog box.

Updating VI Dependencies

To improve performance, the LabVIEW Adapter stores information about VIs it loads. However, if a dependency of a VI is deleted or modified in a way that breaks the main VI, the LabVIEW Module tab does not reflect these changes. You can get the latest information for a VI whose dependencies have changed by unloading all modules.

When a dependency of a VI is modified, unload all modules using one of the following methods:

- In the TestStand Sequence Editor, type <Ctrl+Shift+U>.
- Using the TestStand Engine API, call the `Engine.UnloadAllModules` method.



Note At run time, TestStand displays any errors related to modifications of the dependencies that break the VI.

Related information:

- [LabVIEW Adapter Configuration Dialog Box](#)
- [Engine.UnloadAllModules Method](#)

Calling LabVIEW VIs That Invoke .NET Code in TestStand

TestStand applications like the TestStand Sequence Editor and TestStand C# User Interfaces are .NET Core processes in TestStand 2025 Q2 and later.

Consider the following to ensure functionality when using TestStand to call LabVIEW VIs that invoke .NET code and vice versa:

- TestStand is a .NET Core process, and does not support calling LabVIEW VIs that invoke .NET Framework assemblies in the LabVIEW Runtime Engine. VIs which invoke .NET Framework assemblies can still run if you execute them in the LabVIEW ADE from TestStand.
- TestStand supports calling LabVIEW VIs that invoke .NET Core assemblies in both the LabVIEW Runtime Engine and the LabVIEW ADE.
- Loading TestStand UI controls in LabVIEW may cause issues, as the TestStand engine loads the .NET Core runtime by default and LabVIEW uses a .NET Framework runtime. You can prevent these issues by preventing LabVIEW from loading the .NET Framework:
 - If you are developing a TestStand user interface in LabVIEW:
 1. Open your LabVIEW configuration file at `<LabVIEW>\LabVIEW.ini` and add the key value pair `DisableDotNetFrameworkSupport=True` under the `[LabVIEW]` section. This prevents the LabVIEW ADE from loading the .NET Framework runtime into the process, disables any features that use .NET framework for that session, and allows LabVIEW to load the .NET Core runtime.
 2. Prevent the LabVIEW runtime engine from executing VIs which call into .NET framework assemblies by adding a new section in `LabVIEW.ini` named `[LVRT]` and adding the key value pair `DisableDotNetFrameworkSupport=True` below it.
 3. Enable LabVIEW to load the .NET Core frameworks needed by TestStand by copying the contents of `<TestStand>\AdapterSupport\DotNetSupport\DotnetCore.runtimeconfig.json` and replacing the contents of `<LabVIEW>\LabVIEW.runtimeconfig.json` with the copied data.



Note You must restart LabVIEW for changes to the `LabVIEW.ini` configuration file to take effect.

- If you are executing a TestStand user interface built using LabVIEW or another supported language:
 1. Open the configuration file present next to your executable. For example, in the LabVIEW User Interface shipping example, this would be `TestExec.ini` which is present next to `<TestStand>\UserInterfaces\Full-Featured\LabVIEW\Source Code\TestExec.exe` or `<TestStand Public>\UserInterfaces\Full-Featured\LabVIEW\Source Code\TestExec.exe`.
 2. Prevent the LabVIEW runtime engine loaded by TestStand from loading .NET Framework by adding a new section in your `.ini` file named `[LVRT]` and adding the key value pair `DisableDotNetFrameworkSupport=True` below it.
 3. Add the key value pairs `DisableDotNetFrameworkSupport=True` and `DotNetCoreRuntimeConfigPath=<TestStand>\AdapterSupport\DotNetSupport\DotnetCore.runtimeconfig.json` under the section with the same name as your executable to allow LabVIEW to load the .NET Core runtime and the necessary .NET Core frameworks.



Note TestStand user interface examples possess the necessary configuration file changes to load correctly in LabVIEW.



Note Disabling the .NET Framework is only supported in LabVIEW 2025 and later.

Related reference:

- [Migrating User Interfaces to .NET Core](#)

Executing Source-Only VIs in LabVIEW Runtime Engine

Use the **Version Independent Runtime Engine Support** feature in the LabVIEW Adapter as an option to avoid potential issues when executing source-only VIs using the LabVIEW Runtime Engine through the LabVIEW adapter. These issues can occur if the LabVIEW version the source-only VI was saved or mass-compiled in does not match

the LabVIEW Runtime engine version, as VI version no longer gets updated during a save or mass-compile.

Source-only VI execution errors related to the save version of the VIs can occur when the LabVIEW Runtime Engine version in the LabVIEW adapter is set to `AutoDetect` or set to a specific version. These errors can affect VIs in projects, classes, libraries, and folders containing a `.lvversion` file. Use one of the following methods to prevent these errors when executing source-only VIs:

1. **Option 1:** Disable the **Save version** feature in LabVIEW, and re-save your VIs.
 - a. Close TestStand.
 - b. Open LabVIEW, navigate to **Tools » Options » Environment**, and disable the **Maintain the Save Version of Loaded Projects by Default** feature.
 - c. Clear the compiled object cache in LabVIEW by navigating to **Tools » Advanced » Clear Compiled Object Cache**.
- 

Note If you have files copied to `user.lib` or `instr.lib`, delete the `.vidb` cache file located at `<LabVIEW Install Directory>\VIObjCache\<version>\objFileDB.vidb`.
- d. Set the save version of your LabVIEW projects to **Editor Version** in the project properties.
 - e. Delete any `.lvversion` files you created in your project folders.
 - f. Mass-compile your LabVIEW files.
 2. **Option 2:** Use the **Version Independent Runtime Engine Support** feature in TestStand.
 - a. In TestStand, Enable the **Enable Version Independent Runtime Engine** option in the LabVIEW Adapter Configuration dialog box.
 - b. Set the LabVIEW Runtime **Version** control to match the version of LabVIEW that you used to save or mass-compile your VIs.
 - c. Restart TestStand.
 3. **Option 3:** If your files do not need to be source-only, disable the **Separate compiled code from source file** setting for your VI, project, library, or entire environment in LabVIEW.

Related concepts:

- [Build Source Files and Execute](#)

- [Version Independent Run-Time Engine Support](#)

Related information:

- [Saving for a Previous Version](#)
- [Separating Compiled Code From VIs and Other File Types](#)

Build Source Files and Execute

The **Build Source Files and Execute** feature supports executing source-only VIs in TestStand using the LabVIEW adapter when it is configured to use the LabVIEW runtime engine.

The following options and software are required to use the **Build Source Files and Execute** feature:

- LabVIEW Development System version 2023 Q3 or higher of the same bitness as TestStand must be installed on the machine and must be the active LabVIEW version. Keep the following in mind to ensure that a compatible version of LabVIEW is set as and stays the active LabVIEW version when using the **Build Source Files and Execute** feature:
 - The active LabVIEW version on a machine is the latest version of LabVIEW that was launched on that machine. If a version of LabVIEW Development System is already running, launching a different version does not make the second version the active version. You can make a version of LabVIEW supported by the Build Source Files and Execute option active by closing any open LabVIEW instances and launching the supported version of LabVIEW with the same bitness as TestStand.
 - If a supported version of LabVIEW Development System is not installed or active when TestStand is launched, the Build Source Files and Execute option is disabled. You must make a supported version of LabVIEW Development System active and restart TestStand to enable the feature.
 - Sequence files throw errors on execution if the active LabVIEW version changes to an unsupported version while the Build Source Files and Execute option is enabled. To resolve these errors, make a supported version of LabVIEW Development System active and restart your sequence file execution.



Note When using the **Build Source Files and Execute** feature with LabVIEW 2024 Q3 or later, follow these steps before executing the sequence file:

- Disable the **Maintain the Save Version of Loaded Projects by Default** feature in LabVIEW. This option can be found in the Tools tab in LabVIEW under **Tools » Options » Environment**.
- If you mass-compiled your LabVIEW files before disabling **Maintain the Save Version of Loaded Projects by Default**, set the save version of your LabVIEW projects to **Editor Version** in the project properties.

- The TestStand Tools package must be installed on the machine.



Note This package is installed by default when installing TestStand but is an optional component of the installation.

- The Enable Version Independent Runtime Engine option must be enabled in the LabVIEW Adapter Configuration.



Note Enabling the Build Source Files and Execute option in the LabVIEW Adapter Configuration dialog box will also enable the Enable Version Independent Runtime option. The Enable Version Independent Runtime option cannot be disabled while the Build Source Files and Execute option is enabled.



Note You must restart TestStand for changes related to the Build Source Files and Execute option to take effect.

The LabVIEW runtime engine can only load VIs if compiled code created in a LabVIEW version matching the version of the runtime engine is available. Source-only VIs cannot normally be loaded using the LabVIEW runtime engine, as compiled code for these VIs is only present in LabVIEW's compiled object cache if the VI:

- is opened in the LabVIEW Development Environment and compiled or run, or
- is modified and saved.

The Build Source Files and Execute feature helps avoid this issue when executing source-only VIs using the LabVIEW runtime engine by determining the LabVIEW modules required for execution and building the LabVIEW source files into one or more Packed Project Libraries (PPLs) before execution begins.



Note

- LabVIEW steps which call Express VIs are excluded from the PPL builds.
- LabVIEW modules present in step type sub-steps are not included in PPLs built using the Build Source Files and Execute feature. Custom step type authors should ensure that the LabVIEW modules called from sub-steps of a custom step type have compiled code present to allow execution in the LabVIEW runtime engine.

When using the Build Source Files and Execute feature the LabVIEW modules will load and execute the VIs present in the constructed PPLs during execution instead of the VIs specified in the LabVIEW steps. Since VIs present in PPLs are compiled, they can execute in the LabVIEW runtime engine without issues. Executing the VIs in the built PPLs lets you use the functionality of source-only VIs without requiring compiled code for the VIs to be available.

The Build Source Files and Execute feature builds the LabVIEW source files into PPLs when either:

- You execute the sequence files calling the LabVIEW files for the first time, or
- You make changes to the LabVIEW files being called or the LabVIEW steps calling the files.

TestStand only rebuilds the generated PPLs if at least one LabVIEW module from the sequence file gets loaded before execution. If all the LabVIEW modules from the file are already loaded, any changes made to the source files after that will not be present in the PPLs when running your TestStand sequence.

Edit time features like Edit VI, Create Project and Configure Express VI that are only available when the LabVIEW Adapter is set to LabVIEW Development System are enabled when Build Source Files and Execute feature is enabled.



Note Only one TestStand process can use the Build Source Files and Execute feature at a time. The Build Source Files and Execute feature is disabled for TestStand processes launched while an instance using the feature is already open and remains disabled even after the TestStand process using the feature is closed.

Optimizing PPL Build Times

You can improve PPL build times for source-only VIs by making your LabVIEW files source-only when possible. The compiled code for source-only VIs is stored in LabVIEW's cache and can be used to optimize PPL build times.

To set a VI to source-only, press Ctrl+I to open VI Properties and enable Separate Compiled Code from Source File.

Build Source Files and Execute Constraints

The following features and options are not supported when using the **Build Source Files and Execute** feature:

- The Switching Between VIs and Packed Project Libraries feature is disabled when the **Build Source Files and Execute** feature is enabled. If your sequence file is already configured to switch between VIs and PPLs, you can use the **Switching Between VIs and Packed Project Libraries** feature to use PPL build specifications from your LabVIEW projects instead of a default build specification generated by TestStand.
- The **Preload on File Open** load option does not apply to LabVIEW and Sequence Call steps when using the **Build Source Files and Execute** feature. Steps with this option enabled behave like the load option is set to **Preload Before Execution**.
- The **Load Dynamically** load option does not function for Sequence Call steps when using the **Build Source Files and Execute** feature. Steps with this option selected preloads their modules before execution begins.

Forcing PPL Rebuilds

If you encounter build errors when building LabVIEW source file PPLs in TestStand or want to force a rebuild of existing PPLs, you can clear the Builder cache by selecting

Execute → Clear LabVIEW Builder Cache from the main TestStand window.

Clearing the Builder cache for a sequence file clears the data stored for built PPLs related to that sequence file, forcing the next execution of the sequence file to rebuild PPLs for any LabVIEW VIs called from that file. You can clear the Builder cache for the active sequence file or for all files using previously built PPLs.



Note You can use the **Clear LabVIEW Builder Cache** option to remove any VIs deleted from or skipped in a sequence file from the list of LabVIEW files used in future PPL builds.

Support for Debugging

To debug PPLs built using the **Build Source Files and Execute** option:

1. Enable the **Enable Debugging and Tracing** option to make PPLs built using the **Build Source Files and Execute** feature debuggable.



Note Changes made to this option do not take effect until you restart the TestStand application.

2. Open the source VI you would like to debug, place breakpoints where necessary and save the changes. When the PPL containing the modified VI is built, the corresponding VI in the PPL will also have breakpoints.
3. Execute the sequence you want to debug once to build the PPLs and load VIs into memory in the LabVIEW runtime Engine.
4. Set LabVIEW to use the TestStand executable to debug.
 - a. Open the Debug Application or Shared Library window in LabVIEW by navigating to Operate → Debug Application or Shared Library.
 - b. Attach LabVIEW to TestStand by selecting the following in the Application or Shared Library drop-down menu:
 - SeqEdit.exe - <ProjectName>.lvproj - My Computer to debug VIs called in project context.
 - SeqEdit.exe to debug VIs called in the default Application context.
- Execute your sequence in TestStand. If there are breakpoints present in a called VI,

they will be hit and will break execution to allow for debugging. If there are breakpoints present in called LabVIEW steps, they will also break execution and allow you to step into called VIs to debug using the LabVIEW Development System.



Note Sequences must be run at least once before they can be debugged remotely from the LabVIEW Development System, as the VIs called by the sequence need to be loaded into memory.

Related information:

- [LabVIEW Adapter Configuration Dialog Box](#)

LabWindows/CVI Adapter

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

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

When you specify a module for a step, the TestStand Sequence Editor displays the LabWindows/CVI Module tab and TestStand User Interfaces launch the Edit LabWindows/CVI Module Call dialog box.

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

Related concepts:

- [Code Modules](#)
- [Adapter and Code Module Support for 64-bit TestStand](#)

- [Bitness-Conditional Code for DLLs with Different 32-bit Interfaces and 64-bit Interfaces](#)
- [C/C++ DLL and LabWindows/CVI DLL Support for 64-bit TestStand](#)
- [Calling LabWindows/CVI Code Modules](#)
- [Debugging DLLs](#)
- [Driver Support for 64-bit TestStand](#)
- [Loading Subordinate DLLs](#)
- [Locating the Correct DLL in 32-bit TestStand and 64-bit TestStand](#)
- [Module Adapter Parameter Mapping Guidelines](#)
- [Pointers and Handles in 32-bit TestStand and 64-bit TestStand](#)
- [Programming with the TestStand API in LabWindows/CVI](#)

LabWindows/CVI Step Execution Locations

Use the Step Execution section in the LabWindows/CVI Adapter Configuration dialog box to select where steps execute. The LabWindows/CVI Adapter can run code modules in-process or out-of-process, as the following table describes.

Where Code Modules Execute	Description
In-process	In the same process as the sequence editor or a user interface you are running, without using the LabWindows/CVI development environment.
Out-of-process	In an external instance of the LabWindows/CVI development environment.

The LabWindows/CVI Adapter uses two different copies of the LabWindows/CVI development environment. The adapter uses one copy to execute code modules for out-of-process step execution. By default, the adapter opens `<TestStand Public>\AdapterSupport\CVI\tscvrun.prj` in this copy of LabWindows/CVI. Use this copy of LabWindows/CVI to edit the projects you use to create DLLs, static libraries, and object files.

Related concepts:

- [Out-of-Process LabWindows/CVI Code Module Execution](#)
- [In-Process LabWindows/CVI Code Module Execution](#)
- [Search Directories](#)

Out-of-Process LabWindows/CVI Code Module Execution

To execute tests in an external instance of LabWindows/CVI, the LabWindows/CVI Adapter launches a copy of the LabWindows/CVI development environment and loads an execution server project. You can specify the execution server project to load in the LabWindows/CVI Adapter Configuration dialog box. The default project is `<TestStand Public>\AdapterSupport\CVI\tscvirun.prj`.

When a step calls a function in an object, static library, or DLL file, the execution server project automatically loads the code module and executes the function in an external instance of LabWindows/CVI. When you want a step to call a function in a C source file, you must include the C source file in the execution server project before you run the project. You must also include any support libraries other than LabWindows/CVI libraries the object, static library, or C source file requires.

Debugging Code Modules

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

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

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

Related concepts:

- [Search Directories](#)
- [Code Modules](#)

In-Process LabWindows/CVI Code Module Execution

When executing code modules in the same process as the sequence editor or user

interface, the LabWindows/CVI Adapter loads and runs code modules directly without using the LabWindows/CVI development environment.

Object and Library Code Modules

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

To configure a list of support libraries for the LabWindows/CVI Adapter to load, manually copy the support libraries to the
<TestStand Public>\AdapterSupport\CVI\AutoLoadLibs directory.

Source Code Modules

When TestStand executes code modules in-process, the LabWindows/CVI Adapter cannot directly execute code modules that exist in C source files. Instead, the adapter attempts to find an object file with the same name. When the adapter finds the object file, it executes the code in the object file. When the adapter cannot find the object file, it prompts you to create the object file in an external instance of LabWindows/CVI. When you decline to create the object file, the adapter reports a run-time error.

Debugging DLL Code Modules

You can debug in-process code modules, but the code modules must exist in DLLs enabled for debugging in LabWindows/CVI at the time they were built. To debug a DLL in-process, you must launch the sequence editor or user interface from LabWindows/CVI. Select **Run»Specify External Process** in the LabWindows/CVI project window to identify the executable you want to launch. Select **Run»Debug Project** to launch the executable and begin debugging.

When you click **Step Into** in the sequence editor while the execution is suspended on a step that calls into a LabWindows/CVI DLL you are debugging, LabWindows/CVI suspends on the first statement in the DLL function.

In TestStand 2016 and later, the LabWindows/CVI debugger automatically attaches to a TestStand process when you perform a step into operation on a LabWindows/CVI

Adapter step. If the LabWindows/CVI DLL is debuggable and the source code is available, LabWindows/CVI suspends the process in the called DLL function; otherwise, TestStand performs a step over operation. You do not need to launch the Sequence Editor from LabWindows/CVI to access the debugger or choose to attach the debugger to an existing process.



Note You must be using LabWindows/CVI 2015 SP1 or later for the debugger to automatically attach to a TestStand process.

Refer to the LabWindows/CVI documentation for more information about debugging DLLs.

Related concepts:

- [Code Modules](#)
- [Search Directories](#)

LabWindows/CVI Data Type Equivalencies in TestStand

TestStand provides number, string, enumeration, Boolean, and object reference built-in data types. TestStand also provides several standard named data types, including Path and Error. You can create container data types to hold any number of other data types. TestStand container data types are analogous to C structures in LabWindows/CVI.

LabWindows/CVI includes a greater variety of built-in data types than TestStand does. Appropriate or safe mappings between TestStand data types and LabWindows/CVI data types when calling code modules, are shown in the following table.

LabWindows/CVI C Data Type	TestStand Data Type
char, unsigned char, short, unsigned short, long, unsigned long, float, or double	Number TestStand stores all numeric C data types as double-precision, floating-point numbers. TestStand does not set the format for a number property when assigning a value.
_int64, long long	Number {Signed 64-bit integer}

LabWindows/CVI C Data Type	TestStand Data Type
unsigned __int64, unsigned long long	Number {Unsigned 64-bit integer}
size_t	(32-bit TestStand) Number, (64-bit TestStand) Number {Unsigned 64-bit integer}
const char*, char[], const wchar_t*, const unsigned short*, wchar_t[], or unsigned short[]	Path, String, or Expression
enum	Number or Enumeration
IDispatch *pDispatch, IUnknown *pUnknown, or CAObjHandle objHandle	Object reference
Array of x	Array of TestStand (x)
Pointer to x	Object reference You can store a pointer a LabWindows/CVI code module returns in a TestStand object reference variable. You can then pass a pointer to a subsequent call to a LabWindows/CVI code module.
struct	Container



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

Related concepts:

- [Code Modules](#)

Mapping LabWindows/CVI Parameters

When you load a new code module prototype, the LabWindows/CVI Adapter uses the following guidelines, in order of priority, to copy the configured argument values for parameters from the old prototype to the new prototype.

1. Map arguments based on parameter name and compatible type.
2. Map arguments based on position and compatible type.

Related concepts:

- [Search Directories](#)

Calling Legacy LabWindows/CVI Code Modules

Versions of TestStand earlier than 3.0 required the DLL Flexible Prototype Adapter to call functions in LabWindows/CVI DLLs that did not use a specific prototype. Using TestStand 3.0 or later, you can call functions with a variety of parameter data types, including code modules with legacy function prototypes.

Prototypes of Legacy LabWindows/CVI Code Modules

TestStand supports the following legacy prototypes:

- **Standard legacy prototype:** `void TX_TEST StandardFunc(tTestData *data, tTestError *error)`
- **Extended legacy prototype:** `int TX_TEST ExtendedFunc(const char *params, tTestData *data, tTestError *error)`

In earlier versions of TestStand, National Instruments recommended using the standard prototype. The extended prototype provides compatibility with the LabWindows/CVI Test Executive Toolkit version 2.0 or earlier and offers an additional string parameter.

Although you usually create new code modules using the LabWindows/CVI Module tab for steps that use the LabWindows/CVI Adapter, TestStand can also create legacy-style code modules.

The legacy prototypes contain the **tTestData** and **tTestError** structure parameters, which the LabWindows/CVI Adapter uses to pass values into and out of the code module.

Updating Step Properties

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

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

Before calling a code module, the LabWindows/CVI Adapter assigns values from TestStand to input fields of the **tTestData** structure. After calling the code module, the LabWindows/CVI Adapter copies the values of the output fields of the structures to properties of the step. The LabWindows/CVI Adapter copies a value into a property when the following conditions are true:

- The property exists
- The code module does not change the value of the property directly through the TestStand API

In some cases, the LabWindows/CVI Adapter translates the value of a structure field to a different value in the corresponding property. The following table lists all the properties the LabWindows/CVI Adapter updates and the value translation, if any, the adapter makes.

Structure Member	Valid Values Tests Can Return	Step.Result Property	Step Property Value
result	PASS or FAIL	PassFail	True/False
outBuffer	string value	ReportText	string value
measurement	floating-point value	Numeric	numeric value
stringMeasurement	string value	String	string value
errorFlag	True or False	Error.Occurred	True/False
errorCode	integer value	Error.Code	numeric value
errorMessage	string value	Error.Msg	string value



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

Related concepts:

- [Example Legacy LabWindows/CVI Code Module](#)

Related information:

- [tTestData Structure](#)
- [tTestError Structure](#)

Example Legacy LabWindows/CVI Code Module

When you create a legacy code module for the LabWindows/CVI Adapter, you must add the `stdtst.h` header file located in the `<TestStand Public>\AdapterSupport\CVI` directory to the source file. The `stdtst.h` file includes the type definitions for the **tTestData** and **tTestError** structures.

The following is an example code module that uses the LabWindows/CVI standard prototype:

```
// Simple test example

#include "stdtst.h"

void TX_TEST __declspec(dllexport) FunctionName (tTestData
*testData, tTestError *testError)

{
```

```

int error = 0;

double measurement = 5.0;

char *lastUserName = NULL;

testData->measurement = measurement;

if ((error = TS_PropertyGetValString (testData-
>seqContextCVI, NULL, "StationGlobals.TS.LastUserName", 0,
&lastUserName)) < 0) goto Error;

Error:

// FREE RESOURCES

CA_FreeMemory(lastUserName);

// Set the error flag to cause a run-time error

if (error < 0)

{

testError->errorFlag = TRUE;

testError->errorCode = error;

testData->replaceStringFuncPtr(&testError->

errorMessage, "ErrorText");

}

}

```

Related concepts:

- [Search Directories](#)

- [Updating Step Properties](#)

Related information:

- [tTestData Structure](#)
- [tTestError Structure](#)

C/C++ DLL Adapter

Use the C/C++ DLL Adapter to call C/C++ functions and static C++ class methods in a DLL with a variety of parameter types. You can call global static methods or static class methods in C++ DLLs. You can create the DLL code module with LabWindows/CVI, Microsoft Visual Studio, or any other environment that can create a C/C++ DLL, including LabVIEW-built shared libraries. Use the C/C++ DLL Adapter Configuration dialog box to configure the C/C++DLL Adapter.

When you specify a module for a step the TestStand Sequence Editor displays the C/C++ DLL Module tab on the Step Settings pane to specify a C/C++ DLL Adapter module call, to specify the source code associated with the module call, and to create and edit C/C++ code modules directly from TestStand. TestStand User Interfaces launch the Edit C/C++ DLL Call dialog box.

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

(Windows 10/8.1/7) When you launch Visual Studio from TestStand, Visual Studio runs with the same privileges you used to run TestStand. If you launch TestStand while logged in as a user with standard privileges and then launch Visual Studio from TestStand, you cannot execute tasks in Visual Studio that require administrator privileges.



Note (64-bit TestStand) 64-bit TestStand does not support Measurement Studio C++ MFC data types because no 64-bit version of Measurement Studio C++ MFC exists.

Related concepts:

- [Code Modules](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)
- [Debugging DLLs](#)
- [Features 64-Bit TestStand does not Support](#)
- [Adapter and Code Module Support for 64-bit TestStand](#)
- [C/C++ DLL and LabWindows/CVI DLL Support for 64-bit TestStand](#)
- [Loading Subordinate DLLs](#)
- [Locating the Correct DLL in 32-bit TestStand and 64-bit TestStand](#)
- [Module Adapter Parameter Mapping Guidelines](#)
- [Programming with the TestStand API in Microsoft Visual C++](#)

Exporting Class Methods and Functions in Microsoft Visual Studio

Use the C/C++ DLL Adapter to call functions and static class methods in DLLs. The C/C++ DLL Adapter obtains a list of available functions and methods from the list of exports contained in the DLL file. In addition, the C/C++ DLL Adapter determines parameter type information by reading a type library or by examining the decorated names the Microsoft Visual Studio compiler generates.

To ensure that the functions and methods are available to the C/C++ DLL Adapter, follow these rules when defining the functions and classes:

- Use `__declspec(dllexport)` to export the function or method. You can export an entire C++ class by placing the `__declspec(dllexport)` before the class name, or you can export a single method by placing `__declspec(dllexport)` before the method name.
- Make class methods static and public. The C/C++ DLL Adapter does not allow you to call public non-static methods.
- Use only types the C/C++ DLL Adapter recognizes.
- Avoid using a `.DEF` file to export symbols. Using a `.DEF` file prevents the C/C++ DLL Adapter from obtaining type information for function and method parameters.
- Avoid using the extern "C" syntax to export symbols. The extern "C" syntax prevents the C/C++ DLL Adapter from obtaining type information for function and method parameters.

Refer to the following examples for more information:

Example 1

```
class __declspec(dllexport) ClassName {
public:static void MethodName(void);
};
```

Example 2

```
class ClassName {
public:static void __declspec(dllexport) MethodName(void);
};
```

Example 3

```
void __declspec(dllexport) Function(void);
```

Parsing Parameters from Source Code

When the C/C++ DLL Adapter parses the parameter list in the source code, it must interpret the parameter declarations. Each parameter must have a numeric, array, or object type, and the return value must have a numeric or void type. Some C parameter declarations can be ambiguous, as described in the following table. For example, `char *` and `char []` can each represent either a NULL-terminated string or a fixed-size character array.

C Data Type in Parameter Declaration in Source Code	Parameter Information on Module Tab
<code>char *</code>	Type:C String
<code>char []</code>	Type: C String Buffer Number of Elements: -1
<code>char [nnn]</code> , where <code>nnn</code> is a numeric literal	Type: C String Buffer Number of Elements: <code>nnn</code>
<code>wchar_t *</code>	Type: Unicode String

<code>wchar_t []</code>	Type: Unicode String Buffer Number of Elements: -1
<code>wchar_t [nnn]</code> , where <code>nnn</code> is a numeric literal	Type: Unicode String Buffer Number of Elements: <code>nnn</code>
<code>int *</code>	Type: Signed 32-bit integer Pass: Pointer to value
<code>int []</code>	Type: Array Data Type: Signed 32-bit integer Number of Elements: -1
<code>int [nnn]</code> , where <code>nnn</code> is a numeric literal	Type: Array Data Type: Signed 32-bit integer Number of Elements: <code>nnn</code>



Note The C/C++ DLL Adapter handles the other numeric types in the same way it handles the signed 32-bit integers.

.NET Adapter

Use the .NET Adapter to call .NET assemblies written in any .NET-compliant language, such as C# or Microsoft Visual Basic .NET. TestStand installs .NET if it is not present on your machine during installation, which allows you to take advantage of the capabilities of the .NET Adapter. You can configure the .NET Adapter using the .NET Adapter Configuration dialog box.



Note You can execute .NET Framework assemblies using .NET as long as the resources used in the .NET Framework assemblies are present in .NET.

When you specify a module for a step, the TestStand Sequence Editor displays the .NET Module tab and TestStand User Interfaces launch the Edit .NET Call dialog box.



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

as needed when the prototypes in code modules change.

Related concepts:

- [Code Modules](#)
- [.NET Code Module Support for 64-bit TestStand](#)
- [Adapter and Code Module Support for 64-bit TestStand](#)
- [AnyCPU Assemblies in 32-bit TestStand and 64-bit TestStand](#)
- [Module Adapter Parameter Mapping Guidelines](#)
- [Platform-Specific Assemblies in 32-bit TestStand and 64-bit TestStand](#)
- [Programming with the TestStand API in Microsoft Visual Basic .NET](#)
- [TestStand Sequence Analyzer](#)

Related reference:

- [TestStand 2024 Q4 Changes](#)

Managing Versions of .NET to Use with TestStand

The .NET Adapter uses .NET 8 to call assemblies, and the TestStand Sequence Editor and the example .NET user interfaces target .NET 8.

Managing Versions of .NET to Use with TestStand 2023 Q4 and Earlier

In TestStand 2023 Q4 and earlier, the .NET Adapter uses the .NET Framework Common Language Runtime (CLR) version 4.0 to call assemblies, and the TestStand Sequence Editor and the example .NET user interfaces target .NET Framework 4.0.

By default, an application that uses the .NET Framework Common Language Runtime (CLR) 4.0 cannot load an assembly written in C++/CLI and built against an earlier .NET Framework version. However, you can create an application configuration file to direct the .NET Framework to allow an application to load such an assembly. Place the application configuration file in the same directory as the executable and use a filename with the same name as the executable and append a `.config` extension.



Note An application configuration file is needed only for C++/CLI assemblies that are built against earlier versions of the .NET Framework CLR.

Assemblies created in C# or VB.NET Framework, for example, can be loaded by default, even if the assemblies are built against an earlier version of the .NET Framework CLR.

For example, to specify that the sequence editor can load legacy C++/CLI assemblies, create in the `<TestStand>\Bin` directory a file named `seqedit.exe.config` that contains the following information:

```
<?xml version="1.0"?>

<configuration>

<startup useLegacyV2RuntimeActivationPolicy="true">

<supportedRuntime version="v4.0" />

</startup>

</configuration>
```

When you set the `useLegacyV2RuntimeActivationPolicy="true"` attribute to `true`, you force an application that uses the .NET Framework CLR 4.0 to load all legacy assemblies in the .NET Framework CLR 4.0, including legacy C++/CLI assemblies, but you also disable the ability to run multiple versions of the .NET Framework CLR in-process and side-by-side. Refer to Microsoft MSDN documentation for more information about using this configuration option.

Related concepts:

- [.NET Adapter](#)

Viewing XML Help for .NET Code Modules

If you export XML Documentation files when building .NET code modules and you want TestStand to display help information for a code module in the Sequence Editor, place the XML Documentation file next to the .NET code module loaded by TestStand. If the XML Documentation file is not in the same folder as the .NET code module when TestStand loads the code module, help in the XML Documentation file will not display

in the Sequence Editor.



Note In TestStand 2024 Q4 and earlier, if the .NET code module is installed in the global assembly cache (GAC), TestStand will load the code module on startup from the GAC location, even if you specify the identical .NET code module from a different place on disk.

Related concepts:

- [.NET Adapter](#)

Debugging .NET Assemblies

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

When you use a supported version of Visual Studio and you suspend a sequence on a step that calls a debuggable assembly, click the **Step Into** button in TestStand to suspend Visual Studio at the first statement in the assembly method or property.

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

(Windows 10/8.1/7) When you launch Visual Studio from TestStand, Visual Studio runs with the same privileges you used to run TestStand. For example, if you launch TestStand while logged in as a user with standard privileges and then you launch Visual Studio from TestStand, you cannot execute tasks in Visual Studio that require administrator privileges.



Note

- Save sequence files and workspaces before you stop debugging and

terminate the TestStand process because Visual Studio might terminate the process without prompting you to save modified files in TestStand.

- You must attach Visual Studio to the TestStand process correctly depending on the type of code module you want to debug. For .NET code modules, NI recommends using only managed debugging when attaching Visual Studio.
- When you use LabWindows/CVI to debug a DLL in the TestStand process, you cannot debug a .NET assembly at the same time. When you use Visual Studio to debug an assembly in TestStand and you want to use LabWindows/CVI to debug code modules at the same time, you must configure the LabWindows/CVI Adapter to execute the steps in an external instance of LabWindows/CVI.

Related concepts:

- [Debugging DLLs](#)
- [Options for Stepping Out of a Microsoft Visual Studio Assembly](#)

Related information:

- [TestStand and .NET Framework Compatibility](#)

Enabling Code Module Hot Reloading During Step-Into Debugging in TestStand

TestStand supports hot reloading of .NET code modules during step-into debugging to increase debugging efficiency.

To enable code module hot reloading in TestStand:

1. Create an environment variable named `COMPLUS_ForceENC = 1`.
2. Configure your .NET step to use the Visual Studio Project or Solution where your code is sourced.
3. Step into the configured .NET function from TestStand. This launches Visual Studio, where you can make changes to your .NET function and hot reload the code module once you finalize your changes.



Note Stopping the debugger closes your TestStand application if you are using hot reloading.



Note In TestStand 2025 Q2 and later, hot reloading is not supported in .NET Framework workflows during step-into debugging. You can still attach TestStand to the Microsoft Visual Studio debugger to use hot reloading.

Options for Stepping Out of a Microsoft Visual Studio Assembly

The following table lists the options for stepping out of a Microsoft Visual Studio assembly.

Table 12. Options for Stepping Out of a Microsoft Visual Studio Assembly

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

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

Related concepts:

- [Debugging .NET Assemblies](#)

Managed Code Considerations for TestStand and .NET 8

TestStand 2025 Q2 and later use .NET 8 to run .NET-based tools, utilities, and interfaces. Users working with .NET code in their TestStand projects and sequences should keep the following recommendations and resources in mind when developing their code.

- You should use the `TestStandObjectReleaser` utility class in the `NationalInstruments.TestStand.Utility` DLL under the `NationalInstruments.TestStand.Utility` namespace to handle `TestStand` objects in managed code. This class identifies all `TestStand` objects that can be reached from the root object during construction and ensures proper disposal of these objects in the `Dispose` method. You should call this utility to handle disposal of `TestStand` objects in the `Dispose` method.

```
For protected override void Dispose(bool disposing)
{
    using (new TestStandObjectReleaserExcludingEngineAndUIControls(this))
    {
        if (disposing)
        {
            components?.Dispose();
        }
        base.Dispose(disposing);
    }
}
```

- You should use the `Microsoft.NetCore.App`, `Microsoft.WindowsDesktop.App`, and `Microsoft.AspNetCore.App` .NET runtimes when developing .NET 8-based `TestStand` applications and user interfaces. You can use the below format to include dependencies on these .NET runtimes.

```
<ItemGroup>
<FrameworkReference
    Include="Microsoft.WindowsDesktop.App" />
<FrameworkReference
    Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

- The .NET 8 `Assembly.Load` and `Assembly.LoadFrom` methods both create new `AssemblyLoadContext` (ALC) objects with the `isCollectible` property set to `False` when called, and load the specified assembly in the newly created ALC object. This can cause typecasting and type missing exceptions when calling classes and methods from an assembly, as the same type may be loaded in multiple ALCs. This is especially likely when attempting to cast to a type with an exact, fully qualified type name results in a `Type Mismatch` or `Type Missing` exception. You can avoid this issue by using `AssemblyLoadContext.LoadFromAssemblyPath` to reference the current

ALC instead of `Assembly.Load` or `Assembly.LoadFrom`. The `LoadFromAssemblyPath` method loads the assembly directly into the ALC instance you invoke the method on, preventing you from creating copies of the same assembly in multiple load contexts.

- If you see this error in a code module, you should load your assembly into the TestStand execution ALC. You can accomplish this by getting a reference to the execution ALC using

```
AssemblyLoadContext currentContext =
    AssemblyLoadContext.GetLoadContext(Assembly.GetExecutingAssembly());
```

and then loading your assembly dynamically using `currentContext.LoadFromAssemblyPath()`.

- If you see this error when using .NET Framework modules to load your assembly into the current ALC, use reflection to find the `System.Runtime.Loader.AssemblyLoadContext` like in the below example.

```
Assembly loadedAssembly = null;
try
{
    Assembly assemblyLoadContext =
        Type.GetType("System.Runtime.Loader.AssemblyLoadContext", throwOnError:
            false);
    if (assemblyLoadContext != null)
    {
        MethodInfo getLoadContext = assemblyLoadContext.GetMethod(
            "GetLoadContext",
            BindingFlags.Static | BindingFlags.Public,
            binder: null,
            new[] { typeof(Assembly) },
            modifiers: null);
        if (getLoadContext != null)
        {
            Assembly thisAssembly = Assembly.GetExecutingAssembly();
            object loadContext = getLoadContext.Invoke(null, new[] {
                thisAssembly });
            if (loadContext != null)
            {
                MethodInfo loadFromAssemblyPath =
                    loadContext.GetType().GetMethod(
                        "LoadFromAssemblyPath",
                        new Type[] { typeof(string) });
                if (loadFromAssemblyPath != null)
```

```

        {
            loadedAssembly =
(Assembly)loadFromAssemblyPath.Invoke(loadContext, new[] { fullPath });
        }
    }
}
}
catch
{
    // Ignore any exceptions and just fall back to loading the assembly the
.NET Framework-compatible way
}
loadedAssembly = loadedAssembly ?? Assembly.LoadFrom(fullPath);
return loadedAssembly;

```

ActiveX/COM Adapter

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

When you specify a module for a step, the TestStand Sequence Editor displays the ActiveX/COM Module tab, and the TestStand User Interfaces launch the Edit ActiveX/COM Call dialog box.

Related concepts:

- [Adapter and Code Module Support for 64-bit TestStand](#)
- [ActiveX/COM Code Module Support for 64-bit TestStand](#)
- [Driver Support for 64-bit TestStand](#)
- [Module Adapter Parameter Mapping Guidelines](#)

HTBasic Adapter



Note The HTBasic Adapter is hidden by default. Use the Adapter

Configuration dialog box to configure the HTBasic Adapter.

Use the HTBasic Adapter to call HTBasic subroutines without passing parameters directly to a subroutine. TestStand provides a library of CSUB routines that use the TestStand API to access TestStand variables and properties from an HTBasic subroutine.



Note

- (Windows 8.1/8/7/Vista) HTBasic currently does not support Windows 8.1/8/7/Vista. If you installed the HTBasic 9.0 development environment under Windows 8.1/8/7/Vista, you can perform the Edit Subroutine and Create Subroutine functions on the HTBasic Module tab in the TestStand Sequence Editor when you use a step configured to use the HTBasic Adapter, but the HTBasic code modules might not run correctly.
- (64-bit TestStand) 64-bit TestStand does not support the HTBasic Adapter.

When you specify a module for a step, the sequence editor displays the HTBasic Module tab and TestStand User Interfaces launch the Edit HTBasic Subroutine Call dialog box.

Related concepts:

- [Code Modules](#)
- [Features 64-Bit TestStand does not Support](#)
- [Programming with the TestStand API in HTBasic](#)

Sequence Adapter

Use the Sequence Adapter to pass parameters when you make a call to a subsequence. You can call a subsequence in the current sequence file, in another sequence file, or a sequence file on a remote system. You can also make recursive sequence calls and call sequences asynchronously in a new thread or a new execution. Use the Sequence Adapter Configuration dialog box to configure the Sequence Adapter.



Note By default, the Sequence Adapter does not appear in the Adapter ring control. You can enable the Sequence Adapter by launching the Adapter Configuration dialog box and removing the checkmark from the option in the **Hidden** column. Alternatively, you can use a Sequence Call step.

When you specify a module for a step, the TestStand Sequence Editor displays the Sequence Module tab and TestStand User Interfaces launch the Edit Sequence Call dialog box.

You can use the Sequence Adapter for a step type that can use any module adapter. Using the Sequence Adapter this way is similar to using the built-in Sequence Call step type, except that the Sequence Call step sets the step status to `Passed` instead of `Done` when no failure or error occurs.

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

Use the Variables pane in the Sequence File window to define parameters for a sequence, including the parameter name, the TestStand data type of the parameter, the default value of the parameter, and whether to pass the argument by value or by reference. You can specify a literal value, pass a variable or property by reference or by value, or use the default value the subsequence defines for the parameter.

Related concepts:

- [Adapter and Code Module Support for 64-bit TestStand](#)
- [CPU Affinity Mask Support for 64-bit TestStand](#)
- [Module Adapter Parameter Mapping Guidelines](#)
- [Remote Execution in 32-bit TestStand and 64-bit TestStand](#)
- [Sequence Call Adapter Support for 64-bit TestStand](#)

Related information:

- [Sequence Call Step](#)
- [Sequence Adapter Configuration Dialog Box](#)

Python Adapter

The Python Adapter provides advanced functionality for calling Python code modules from TestStand.

Before using the Python Adapter in TestStand, install the required version of the CPython interpreter.



Note During installation of the CPython interpreter, you must enable the option for adding the Python path to the environment variables.

Install the following Python packages to enable specific workflows:

Package	Version	Workflow
Python for Win32 (pywin32)	228	Passing TestStand objects or ActiveX objects that implement IDispatch between TestStand and Python.
Numpy	1.19.1	Passing an array of numbers from TestStand to Python as a NumPy array.
debugpy	–	Perform step into debugging operations on Python modules.
Jedi	–	View additional elements used in Python code modules, including inner classes, function and attribute names defined in a base class, and imported classes, attributes, and functions.

You can use the Python Adapter to complete the following tasks:

- Execute Python scripts (.py) on disk by:
 - Calling a function defined in a module.
 - Getting/Setting the attributes defined in a module.
 - Creating a class instance.
 - Calling a member function or static function defined in the class.
 - Getting/Setting member attributes or static attributes defined in the class.

- Execute Python scripts in supported Python versions. Refer to the TestStand Release Notes for supported Python versions.
- Execute Python scripts out-of-process in the CPython interpreter.
- Convert data between Python and TestStand variables.
- Store/reuse the Python object in a TestStand variable (Object Reference).

You can also use multiple Python interpreter sessions to perform the following tasks:

- Execute Python scripts in parallel.
- Use multiple Python versions, such as 3.6 and 3.8, simultaneously in TestStand.

When you specify a module for a step, the TestStand Sequence Editor displays the Python Module Tab. TestStand User Interfaces launch the Edit Python Call dialog box.

Use the Python Adapter Configuration Dialog Box to configure the Python version and Python virtual environment path to use with the Python Adapter.

Related information:

- [Python Download](#)
- [Edit Python Call Dialog Box](#)
- [Python Adapter Configuration Dialog Box](#)

Python Adapter Known Limitations

The following are the known limitations when using Python Adapter:

- You cannot use Python interpreters other than CPython, such as PyPy.
- You cannot pass .NET objects between Python and TestStand.
- You cannot call a function in a class with the @classmethod attribute.
- You cannot log a return value unless it is stored in a TestStand variable.
- You cannot use default values. Values for all the parameters of a Python function need to be specified.
- You cannot pass parameters using a keyword (or named) argument.
- You cannot convert an empty container, or a container that does not have any data, between TestStand and Python.
- You cannot convert an Array of unstructured containers from TestStand to Python.
- You cannot use TestStand to create or edit Python code.

Loading and Unloading Python Modules

The **Load Option** setting of the step will execute when Python module is configured to use the Global interpreter session created per instance of TestStand. Python modules are loaded dynamically for all other Global interpreter session options.

The **Unload Option** setting for the step using Python modules cannot be used. Once the Python modules are loaded in a Global interpreter session, they cannot be unloaded.

Handling Python Code Execution when Importing Modules

Include conditional code in your Python modules to handle cases where a module can function as the top-level environment or an imported module.

Python is designed in such a way that when the top-level Python module imports a second Python module, the interpreter evaluates and executes the statements in the second module. With the Python Adapter in TestStand, the interpreter treats TestStand as the top-level environment. TestStand imports the specified module at certain events to determine if the module is valid or not, and to determine the contents of the module. Therefore, any Python module you specify in a Python Adapter step will run when TestStand loads the module, even when the sequence is not executing.

To prevent the interpreter from executing Python code when it imports a module, encapsulate Python module code within the following statement:

```
if __name__ == "__main__":
```

When a Python module runs directly as the top-level script, the interpreter sets the variable `__name__` to `__main__`. For imported modules, the interpreter sets the variable `__name__` to the name of the module. Encapsulating executable code within the conditional statement ensures that the interpreter will check whether the module code is running in the `__main__` environment, and if not, the code will not execute on import.

Refer to Python documentation for more information about using the `__name__` variable.

Related information:

- [Module Tab - Edit Python Call Dialog Box](#)
- [Python Parameter Passing Tab - Type Properties Dialog Box](#)

Python Interpreter Sessions

You can use the Advanced Settings Dialog to specify which Python interpreter session to use for executing the step. You can specify one of the following options:

- **Object Reference**—Use the interpreter session present in interpreter reference
- **Per Thread**—Use the interpreter session created per TestStand thread
- **Per Execution**—Use the interpreter session created per TestStand execution
- **Global**—Use the interpreter session created once per TestStand application

You can use multiple Python interpreter session to execute Python scripts in parallel or use multiple Python versions simultaneously.

Execute Python Scripts in Parallel

CPython is one of the most used Python interpreters. It has a Global Interpreter Lock (GIL) due to which only one Python thread can be executing at any given point of time. Hence, you cannot perform any multi-threaded operation in Python to execute operations in parallel.

National Instruments recommends you to use multiple Python interpreter session to execute Python modules in parallel. If you want to handle device instruments using Python, you should create an interpreter session for each instrument so that all instruments can be operated in parallel.

Use Multiple Python Versions Simultaneously

You can also use multiple Python interpreter sessions to use different Python versions together in the same TestStand execution.



Note A Python object created in a Python interpreter session and stored in a TestStand variable of type object reference should only be used in that

session.

Console for Interpreter Sessions in Python

You can use Advanced Settings Dialog to specify if you want to display console for each Python interpreter session created for executing Python code modules.

The console will be automatically closed based on the lifetime of Python interpreter session which is configured in **Python Interpreter to Use**. The console window for the Global interpreter option will stay open between executions. Call 'Unload All Modules' to properly unload the Global interpreter and close the window.

Manually closing the console will close the Python interpreter session which might lead to error when executing Python code module. Hence, you should not close Python interpreter session console.



Note You cannot access the data, printed in console, from TestStand. If you need any such data, you need to return the data to TestStand from a function call or store the data in an attribute and access the attribute from TestStand.

Python Namespaces

Namespaces enable TestStand to load multiple Python modules with the same name from different locations, or to assign the full type name to a class.

When you specify a Python module for the Python Adapter to use, TestStand assigns a namespace to the module. If the module you specify is under a package, TestStand calculates the namespace from the top most package. If the module you specify is not under a package, the namespace contains only the module name.

Compatibility

In earlier versions of TestStand before namespace support was introduced, type names of classes did not contain namespaces. If you use the `type()` function to determine the data type of a parameter of a function, you may encounter errors that were not present when using the earlier version of TestStand. Update your type validation code to resolve such errors.

Related information:

- [Packages in Python](#)

Debugging Python Modules

You can perform step into debugging operations on Python modules using the Python Adapter.

Before you can step into Python modules, you must install the following software and configure the Python Adapter:

1. Install the debugpy package in Python.
2. Install Visual Studio Code. Select either User Installer or System Installer from the download page. TestStand uses VS Code as the IDE for debugging Python modules.
3. Install NI TestStand Support for Debugging Python Code. This VS Code extension also installs the Python extension if your computer is connected to the internet. If the computer is not connected to the internet, you must install the Python extension for VS Code.

If your computer had VS Code installed before installing TestStand, the TestStand installation process also installs the VS Code extension. In VS Code, confirm that NITestStand Support for Debugging Python Code is installed. Otherwise, complete the following steps:

- a. Launch the TestStand Version Selector.
- b. Under Installed TestStand Versions, select TestStand 2021 SP1 or later, then click **Activate** or **Configure**. The TestStand Version Selector performs the action of installing the extension when you activate TestStand 2021 SP1 or later.



Note If you ever uninstall the extension, repeat this step to reinstall it.

4. Configure Workspace Trust in VS Code.
 - a. Click **Manage»Workspace Trust**.
 - b. Click **Add Folder**.
 - c. Add `C:\<My Documents>\TestStand\Python` as a trusted folder.



Note You may need to create the folder first.

5. Configure the Python Adapter for Debugging.
 - a. In the TestStand Sequence Editor, click **Configure»Adapters**, select **Python**, and click **Configure**.
 - b. Select the **Enable Debugging** option.
 - c. Select **Enable Just My Code** to debug only your Python code and exclude installed libraries when debugging. Otherwise, deselect **Enable Just My Code** to debug both your Python code and any installed libraries your code imports.
 - d. For the **Executable Path** option, browse to the location of `python.exe` you want to use to debug Python modules. The version number of the executable must match the version number in the **Version** option.

Complete the following steps to debug functions you call from TestStand using the Python Adapter.

1. Place a breakpoint at a step that calls a Python module.
2. Select **Execute»Run MainSequence**. The execution pauses at the Python step.
3. Click the **Step Into** button at the top of the Execution window. VS Code becomes the active application, in which the Python module is open and in a suspended state.
4. Click **Step Into** or **Step Over** on the VS Code toolbar to begin stepping through the function. You can click **Continue** at any time to finish stepping through the module and return to TestStand.
5. Click the **Resume** button on the Execution toolbar in TestStand to complete the execution.

Known Limitations of Python Step-Into Debugging

The following considerations are important to keep in mind when debugging Python modules.

- Sometimes VS Code will not come to front when you step into a Python module, but it will blink in the taskbar.
- You cannot step into set/get attributes.
- You cannot step into Python code if the step overrides Python Adapter settings.
- During debugging, if you come out from VS Code to TestStand by clicking step into

or step over, VS Code stops when any additional Python code is executed, regardless of whether you step into that Python code from TestStand.

Attaching the Python Debugger to an External Process

Python code modules are executed in an external process on the system where the Python interpreter is hosted. You can attach the Python debugger to the external process.

Use the following code as the first line in the function that is being called from the Python Adapter to attach the debugger to the Python code module.

Python Version	Code
3.6+	<pre>import os import ctypes ctypes.windll.user32.MessageBoxW(None, "Process name: niPythonHost.exe and Process ID</pre>

A dialog box with the process name is displayed upon execution. Attach the Python debugger to the specified process and click **OK** to start debugging the code module in your debugger.

Updating Python Modules while Debugging

You can modify and reload Python modules while debugging a test sequence.



Note The following steps are useful during test development and debugging. However, NI recommends you avoid reloading updated Python modules when running tests in a production environment.

1. Ensure that **Reload Modified Modules Before Execution** is enabled in the Python Adapter Configuration dialog box.



Note TestStand prompts you to unload all modules when you modify this setting.

2. Execute the sequence and step into a Python code module.
3. While execution is stopped, modify the code module the Python step calls, then resume execution. TestStand will use any modifications you make when it executes the step.

The following are the known limitations when updating modules while debugging:

- Reloading an updated module behaves like importing the module for the first time. Global variables are reset to the default value, and any statement at the module level is executed.
- TestStand reloads only the module the step executes, but not dependent modules.
- If you modify a module that defines a class and another module stores an instance of that class, TestStand does not reload the second module when it reloads the first one. You must reload the second module to discard any instances of the objects and create new instances.
- If any Python object such as a class instance exists, and you modify the object, you must recreate the class instance after reloading the module.

Database Logging

TestStand can automatically log results from executions to a database and provides built-in step types for accessing databases directly from test sequences.

The database logging capability is not native to the TestStand Engine or the TestStand Sequence Editor. The `NI_DatabaseLogger.seq` process model plug-in implements the database logging features. You can also customize or replace any portion of the database logging plug-in.

The default process model, which calls the Main sequence in the client sequence file to test a UUT, relies on the automatic result collection capabilities of the engine to accumulate the raw data to log to a database for each UUT. The engine automatically compiles the result of each step into a result list for an entire sequence, which contains the result of each step and the result list of each subsequence call it makes.

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

Select **Configure»Result Processing** to launch the Result Processing dialog box. Click the checkbox in the Enabled column of the Database row to enable database logging. Click the icon in the Options column to launch the Database Options dialog box, in which you can specify the following options:

- The data link connection string TestStand uses to log results.
- The database schema TestStand uses. A schema contains the SQL statements, table definitions, and TestStand expressions that instruct TestStand how to log results to a database. TestStand includes a set of predefined schemas, which contains at least one schema for each supported database management system. You can also create new schemas that log results to tables you define.
- Filtering options to limit the amount of data TestStand logs.
- If the process models log data after executing each step or after passing through each UUT loop.

Related concepts:

- [Process Model Plug-In Architecture](#)
- [Process Model Architecture](#)
- [Result Collection](#)
- [Configuring Logging to Custom Database](#)
- [TestStand Database Fundamentals](#)

TestStand Database Fundamentals

To use database logging in TestStand, you must have a working knowledge of database concepts, SQL, and database management system (DBMS) client software.

Review the following key concepts for using databases with TestStand and the following key Microsoft Windows features TestStand uses to communicate with a database management system (DBMS).

Databases and Tables

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

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

UUT_NUM	STEP_NAME	RESULT	MEAS
20860B456	TEST1	PASS	0.5
20860B456	TEST2	PASS	(NULL)
20860B123	TEST1	FAIL	0.1
20860B789	TEST1	PASS	0.3
20860B789	TEST2	PASS	(NULL)

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

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

Database Sessions

Database operations occur within a database session. The lifetime of the session varies depending on the component connection to the database. Use the Database results processing plug-in to log execution results to a database. Use the Property Loader step type to import property and variable values from a file or database during an execution. Use a code module or the built-in Database step types to communicate with a database.

A simple database operation includes the following steps:

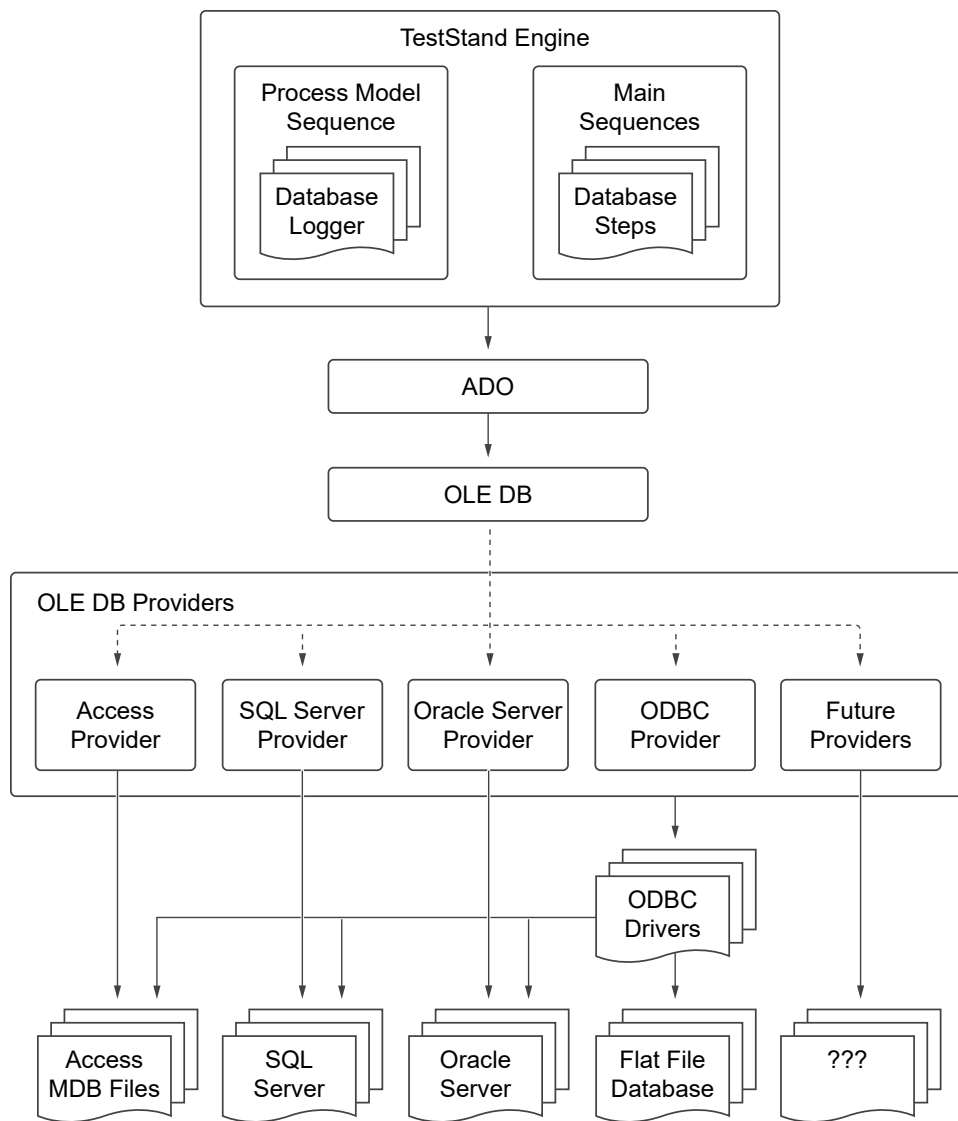
1. Use the Open Database step type to connect to a database.
2. Use the Open SQL Statement step type to perform an SQL query on tables in the database.
3. Use Data Operation step types to create new records and to retrieve and update existing records.
4. Use the Close SQL Statement step type to close the SQL query.
5. Use the Close Database step type to disconnect from a database.

Right-click the Database step and select **Edit <step type>** from the context menu to configure the step type and to set the custom step properties. You can also click the **Edit <step type>** button on the edit tab of the Step Settings pane.

Microsoft ADO, OLE DB, and ODBC Database Technologies

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

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



Refer to the Microsoft website at www.microsoft.com for more information about database technologies for Windows operating systems.

Related information:

- [Microsoft Home Page](http://www.microsoft.com)

Structured Query Language (SQL)

The Structured Query Language (SQL) is a widely supported standard for database access. You can use SQL commands to manipulate rows and columns in database tables in TestStand.

The following SQL commands are commonly used for database table manipulation in TestStand:

- CREATE TABLE
- SELECT
- INSERT
- UPDATE
- DELETE

Consult the SQL API reference at www.w3schools.com/sql/default.asp for information on the usage of these and other SQL commands.

Related information:

- [W3 SQL API Reference](#)

Format Strings for Database Date Values

The Format control on the Column/Parameters section of the Database Options dialog box and the Format String control on the Column/Parameter Values section of the Edit Data Operation dialog box support the following format strings when reading date values from a database and writing date values to a database. A format string consists of symbols that describe how to convert a date value read from a database to a string value, or how to interpret a string value when writing a date value to a database.

Format String	Value	Formatted Value
mm/dd/yy	Oct 20, 2010	10/20/10
dd.mm.yy	Oct 20, 2010	20.10.10
'Jane Doe, born' Mmmm d, yyyy	Oct 20, 2010	Jane Doe, born October 20, 2010
hh:mm:ss	3:47:42 PM	15:47:42
hh:mm:ss AM/PM	3:47:42 PM	03:47:42 PM

Symbol	Description	Example Output
m	Month as number without leading zero.	12, 5
mm	Month as number with leading zero when applicable.	12, 05
mmm	Month as three-letter abbreviation, lowercase.	mar
Mmm	Month as three-letter abbreviation, initial cap.	Mar
MMM	Month as three-letter abbreviation, uppercase.	MAR
mmmm	Month as full name, lowercase.	march
Mmmm	Month as full name, initial cap.	March
MMMM	Month as full name, uppercase.	MARCH
d	Day of the month as number without leading zero.	25, 5
dd	Day of the month as number with leading zero when applicable.	25, 05
ddd	Day of the month as three-letter abbreviation, lowercase.	tue
Ddd	Day of the month as three-letter abbreviation, initial cap.	Tue
DDD	Day of the month as three-letter abbreviation, uppercase.	TUE
dddd	Day of the month as full name, lowercase.	tuesday
Dddd	Day of the month as full name, initial cap.	Tuesday
DDDD	Day of the month as full name, uppercase.	TUESDAY
YY	Last two digits of year.	08
YYYY	Four-digit year.	2010
h	Hour of the day, without leading zero (use am/pm symbol for 12-hour style).	12, 5
hh	Hour of the day, with leading zero (use am/pm symbol for 12-hour style).	12, 05
i (or m)	Minute of the hour, without leading zero.	57, 5
ii (or mm)	Minute of the hour, with leading zero.	57, 05
s	Second of the minute, without leading zero.	57, 5
ss	Second of the minute, with leading zero.	57, 05
ss.ssssss	Second of the minute with fractional seconds (up to six 's' symbols)	57.123456

Symbol	Description	Example Output
	after the decimal point).	
am/pm	"am" or "pm" string, lowercase (forces 12-hour clock).	am
AM/PM	"AM" or "PM" string, uppercase (forces 12-hour clock).	AM
a/p	"a" or "p" string (forces 12-hour clock).	a
A/P	"A" or "P" string, uppercase (forces 12-hour clock).	A
/ - . : , <space>	Output the character.	—
\<character>	Output the character following the '\' character.	\U\T\C is UTC
"<string>" '<string>'	Output the string.	"UTC" is UTC
GD	General format for dates is the Short Date Format in the Regional Options section of the Microsoft Windows Control Panel.  Note Do not combine other format symbols with GD except [US].	—
GDT	General format for dates with times. The Time Format control in the Regional Options section of the Windows Control Panel is appended to the Short Date Style. This is the default when no format string is given.  Note Do not combine other format symbols with GDT except [US].	—
GL	General long format for dates. The Long Date Style control in the Regional Options section of the Windows Control Panel.  Note Do not combine other format symbols with GL except [US].	—
GLT	General long format for dates with times. The Time Style control in the Regional Options section of the Windows Control Panel is appended to the Long Date Format.	—

Symbol	Description	Example Output
	 Note Do not combine other format symbols with <code>GLT</code> except <code>[US]</code> .	
GT	General format for time. The Time Style control in the Regional Options section of the Windows Control Panel.  Note Do not combine other format symbols with <code>GT</code> except <code>[US]</code> .	—
[US]	Combine with <code>GD</code> , <code>GDT</code> , <code>GL</code> , <code>GLT</code> , or <code>GT</code> to override the Regional Options section of the Windows Control Panel and use the United States defaults instead.	—

Data Links

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

For example, to connect to a Microsoft SQL Server database, specify the Object Linking and Embedding Database (OLE DB) provider for an SQL Server, a server name, a database name, a user ID, and a password. To connect to a Microsoft Access database, specify Microsoft Jet in 32-bit TestStand and Microsoft ACE OLE DB in 64-bit TestStand or specify the OLE DB provider for Open Database Connectivity (ODBC) and an ODBC data source name. The ODBC data source name specifies which ODBC driver to use, the database file (`.mdb`), and an optional user ID and password. Use the ODBC Administrator on the Windows Control Panel to define ODBC data source names.



Note Refer to the documentation for the database you use to determine whether and how you can communicate to the database securely. NI recommends that you do not use a plain text password to connect to databases with sensitive information.

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

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

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

1. Create a Data Link file by right-clicking in Windows Explorer and selecting **New»Text Document**.
2. Change the file extension to .udl.
3. Right-click the new file and select **Open** to launch the Data Link Properties dialog box.
4. Click the Connection tab and enable the **Use connection string** option.
5. Click the **Build** button to launch the Select Data Link dialog box, in which you can build a connection string. When you finish, click **OK** to close the Select Data Link dialog box.
6. Use the other options on the Provider, Connection, Advanced, and All tabs to provide additional configuration information for the .udl file.
7. Click **OK** to close the Data Link Properties dialog box. The .udl file automatically saves when you exit the dialog box.

Related concepts:

- [Data Links](#)

Example Data Link and Result Table Setup for Microsoft Access

You can link a TestStand data link to a Microsoft Access database file (.mdb) using the Jet Object Linking and Embedding Database (OLE DB) provider in 32-bit TestStand and the Microsoft Office 12.0 Access Database Engine OLE DB Provider in 64-bit TestStand to log results using the default process model. Visit ni.com/info and enter the Info

Code 64TSaccdb to access the NI support article, Using Microsoft Access Databases with 64-bit TestStand, for more information about installing this provider.

Related concepts:

- [Database Options—Specifying a Data Link and Schema](#)
- [Microsoft ADO, OLE DB, and ODBC Database Technologies](#)

Database Options—Specifying a Data Link and Schema

Complete the following steps to configure the database logging options.

1. Launch the TestStand Sequence Editor and log in as Administrator.
2. Select **Configure»Result Processing** to launch the Result Processing dialog box and click the icon in the Options column for the built-in database plug-in to launch the Database Options dialog box. The Logging Options tab is active.
3. Enable database logging by removing the checkmark in the **Disable Database Logging** option.
4. Click the Data Link tab of the Database Options dialog box.
5. Select **Access** from the Database Management System ring control.
6. Click the **Build** button to launch the Data Link Properties dialog box.
7. Select **Microsoft Jet 4.0 OLE DB Provider** on the Provider tab of the Data Link Properties dialog box if the 32-bit version of the application is running. If the 64-bit version of the application is running, select the **Microsoft Office 12.0 Access Database Engine OLE DB Provider** Visit ni.com/info and enter the Info Code 64TSaccdb to access the NI support article, Using Microsoft Access Databases with 64-bit TestStand, for more information about installing this provider..
8. Click **Next**.
9. On the Connection tab, click **Browse** to launch the Select Access Database dialog box. (32-bit TestStand) Using the Select Access Database dialog box, locate a Microsoft Access database file (.mdb) and click **Open** to select the file. (64-bit TestStand) Enter the path of the Microsoft Access database file (.mdb) in the Data Source control.
10. In the Data Link Properties dialog box, click the **Test Connection** button to verify that you properly entered the required information.
11. Click **OK** to close the Data Link Properties dialog box.

Notice that the Connection String Expression in the Database Options dialog box now

contains a literal string expression version of the data link connection string.

Database Viewer—Creating Result Tables

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

1. If you are continuing from the steps in the Database Options—Specifying a Data Link and Schema section, skip to step 2. Otherwise, complete the following steps.
 - a. Launch the TestStand Sequence Editor and log in as `Administrator`.
 - b. Select **Configure»Result Processing** to launch the Result Processing dialog box and click the icon in the Options column for the built-in database plug-in to launch the Database Options dialog box. The Logging Options tab is active.
 - c. Enable database logging by removing the checkmark in the **Disable Database Logging** option.
 - d. Click the Data Link tab of the Database Options dialog box.
2. Click the **View Data** button to launch the Database Viewer application and open the data link.



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

3. In the Database Viewer application, select **File»Execute SQL Window** to open an Execute SQL window.
4. Click the **Load From SQL File** button.
5. Select `<TestStand>\Components\Models\TestStandModels\Database\Access Create Generic Recordset Result Tables.sql` and click **Open**. Notice that the SQL Commands control now contains a set of SQL commands for creating the default result tables.
6. Click the **Execute** button to create the default result tables.
7. Review the results of the SQL commands on the Output tab to ensure that you created the tables successfully.
8. Click the **Refresh** button in the Database Explorer pane to view the created tables.

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

Related concepts:

- [Database Options—Specifying a Data Link and Schema](#)
- [TestStand Directory Structure](#)
- [Entry Points](#)

Configuring Logging to Custom Database

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

1. Ensure that you thoroughly understand fundamental database concepts.
2. Decide which database management system (DBMS) you want to use. By default, TestStand supports SQL Server, Oracle, Microsoft Access, Sybase, and MySQL. You can add support for other database management systems.
3. Ensure you installed the appropriate client DBMS software required to communicate with the DBMS. You must decide to use an Open Database Connectivity (ODBC) driver or a specific Object Linking and Embedding Database (OLE DB) provider for the DBMS. Use the OLE DB providers for SQL Server and Access. Most Oracle ODBC drivers and OLE DB providers require that you install Oracle Client. Refer to the Recommended Database Client Software section of the NI TestStand Readme for more information about suggested providers, versions of ODBC drivers, client DBMS software, and any known issues.
4. Create the default database tables in the DBMS. The `<TestStand>\Components\Models\TestStandModels\Database` directory contains SQL script files to create and delete the default database tables the default TestStand schemas require. For example, the `Access Create Generic Recordset Result Tables.sql` file contains SQL commands to create the default tables for Access. The `Access Drop Result Tables.sql` file contains SQL commands to delete the default tables. TestStand includes an example Access database, `TestStand Results.mdb`, in the `<TestStand>\Components\Models\TestStandModels\Database` directory. You can also create the default database tables using an SQL script file.
5. Use the Database Options dialog box to enable database logging and to define a data link and schema for the default process model to use.

Related concepts:

- [TestStand Database Fundamentals](#)

- [Adding Support for Other Database Management Systems](#)
- [TestStand Directory Structure](#)
- [Default TestStand Table Schemas](#)
- [Database Viewer—Creating Result Tables](#)
- [Data Links](#)
- [Database Known Issues](#)

Connecting to a Database with the ODBC Administrator

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

(Windows 10/8.1/7) If you use the ODBC Administrator and you receive any warnings, follow the prompts. You might need administrator access to create a new Database Source Name (DSN).



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

Related concepts:

- [Microsoft ADO, OLE DB, and ODBC Database Technologies](#)
- [ODBC Data Source Administrator Dialog Box](#)

ODBC Data Source Administrator Dialog Box

Navigate to the standard Microsoft Windows Control Panel facility for administrative

tools and select **Data Sources (ODBC)** to launch the ODBC Data Source Administrator dialog box.

The ODBC Data Source Administrator dialog box lists all the registered ODBC data sources. The dialog box contains tabs for defining data sources and viewing available drivers.

- **User DSN**—Defines data sources visible only to you.
- **System DSN**—Defines data sources for all users.
- **File DSN**—Specifies that the ODBC Administrator stores the data source definitions in a directory you specify.
- **Drivers Tab**—Displays the available drivers for the system.

For the User DSN and System DSN tabs, the ODBC Administrator stores the data source definitions in the Windows system registry. Click **Add** or **Configure** on these tabs to launch a driver-specific dialog box, in which you can configure a new or an existing data source. The system then saves the configuration for the data source in the registry or as a file.

Click **Help** to launch the Microsoft online help for any tab in the ODBC Data Source Administrator dialog box.

TestStand Database Result Tables

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

Creating Default Result Tables with the Database Viewer

Use the TestStand Database Viewer to create the default result tables the schema requires. You can also use the Database Viewer application to view data in a database, edit table information, and execute SQL commands.



Note You must set up the database management system (DBMS) server and any required DBMS client software before using the Database Viewer application.

Related concepts:

- [Default TestStand Table Schemas](#)
- [Adding Support for Other Database Management Systems](#)
- [Data Links](#)
- [Database Viewer—Creating Result Tables](#)
- [Logging Attribute Values](#)

Default TestStand Table Schemas

The default TestStand database schema requires the following tables in the database:

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

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

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



Note

- The default TestStand database schemas do not log attribute values for result properties.

- The default database schemas assume that result tables conform to the table definitions. When you want to modify the table schema, you must alter the tables in the database and create a new schema using the Database Options dialog box.

Related concepts:

- [UUT_RESULT Table Schema](#)
- [STEP_RESULT Table Schema](#)
- [STEP_SEQCALL Table Schema](#)
- [PROP_RESULT Table Schema](#)
- [PROP_BINARY Table Schema](#)
- [PROP_ANALOGWAVEFORM Table Schema](#)
- [PROP_DIGITALWAVEFORM Table Schema](#)
- [PROP_NUMERICLIMIT Table Schema](#)
- [Logging Attribute Values](#)

Using Legacy Database Schemas

Configure TestStand to use a legacy database schema when logging results from executions to a database.

1. Click **Configure»Result Processing** to launch the Result Processing dialog box.
2. In the Database row, click the **Options** icon to launch the Database Options dialog box
3. Select the **Schemas** tab, click **Reload NI Schemas**, then select the legacy schema you want to use.

Related concepts:

- [Updates to Default Database Schemas](#)
- [Default TestStand Table Schemas](#)

UUT_RESULT Table Schema

The default TestStand schema logs station, batch, UUT, and execution information to

this table.

Table 13. UUT_RESULT Table Schema

Column Name	Data Type	Description
ID	Primary Key	Unique value that identifies each entry in the table. For Access and SQL Server, the default process model assumes this column increments automatically. For Oracle, the default process model uses an Oracle SQL sequence to generate a unique number.
STATION_ID	String	Station ID, usually the computer name.
BATCH_SERIAL_NUMBER	String	Serial number of the batch. This applies only to executions that use the Batch process model.
TEST_SOCKET_INDEX	Number	Test socket for the UUT. This applies only to executions that use the Parallel or Batch process models.
UUT_SERIAL_NUMBER	String	Serial number of the UUT.
USER_LOGIN_NAME	String	Login name of the user who tested the UUT.
START_DATE_TIME	Date-time	Time and date at which the UUT test began executing.
EXECUTION_TIME	Number	Number of seconds the UUT test took to execute.
UUT_STATUS	String	Status of the UUT test.
UUT_ERROR_CODE	Number	Error code when the UUT test status is <code>Error</code> .
UUT_ERROR_MESSAGE	String	Error message when the UUT test status is <code>Error</code> .
PART_NUMBER	String	Part number for the UUT.
TSR_FILE_NAME	String	Name of the offline results log file that created the UUT record.
TSR_FILE_ID	String	ID of the offline results log file that created the UUT record.
TSR_FILE_CLOSED	Boolean	Indicates whether the offline results file was closed properly.

STEP_RESULT Table Schema

The default TestStand schema logs common subproperties of all steps to this table.

Table 14. STEP_RESULT Table Schema

Column Name	Data Type	Description
ID	Primary Key	Unique value that identifies each entry in the table. For Microsoft Access and Microsoft SQL Server, the default process model assumes that this column increments automatically. For Oracle, the default process model uses an Oracle SQL sequence to generate a unique number.
UUT_RESULT	Foreign Key	Unit Under Test (UUT) ID from the UUT_RESULT table that associates the step result with a UUT.
STEP_PARENT	Foreign Key	ID of the parent sequence call step result when the step is a step in a subsequence.
STEP_NAME	String	Name of the step.
STEP_TYPE	String	Name of the step type.
STEP_GROUP	String	Step group that contains the step. The value is "Main", "Setup", or "Cleanup".
STEP_INDEX	Number	Zero-based position of the step in the step group.
STEP_ID	String	Unique ID of the step.  Note This Step_ID property is not the same as the TS.StepId used to report test results.
STATUS	String	Status of the step.
REPORT_TEXT	String	Report text of the step.
ERROR_CODE	Number	Error code when the step status is <code>Error</code> .
ERROR_MESSAGE	String	Error message when the step status is <code>Error</code> .
CAUSED_SEQFAIL	Boolean	Indicates whether the step failure caused the sequence to fail.
MODULE_TIME	Number	Number of seconds the step module took to execute.

Column Name	Data Type	Description
TOTAL_TIME	Number	Number of seconds the step took to execute, including module execution and all step options, such as preconditions, expressions, post actions, and module loading.
NUM_LOOPS	Number	Number of loops the step executed, if any.
NUM_PASSED	Number	Number of loops the step returned with a status of <code>Passed</code> , if any.
NUM_FAILED	Number	Number of loops the step returned with a status of <code>Failed</code> , if any.
ENDING_LOOP_INDEX	Number	The loop index after executing the ending loop.
LOOP_INDEX	Number	The value of the loop index for an iteration of the step.
INTERACTIVE_EXENUM	Number	Number TestStand assigns to an interactive execution. The number is unique with respect to all other interactive executions in the current TestStand session. TestStand adds this property only when you run the step interactively.
ORDER_NUMBER	Number	The order in which the step executed within the execution.

STEP_SEQCALL Table Schema

The default TestStand schema logs subproperties of the Sequence Call step to this table.

Table 15. STEP_SEQCALL Table Schema

Column Name	Data Type	Description
ID	Primary Key	Unique value that identifies each entry in the table.
STEP_RESULT	Foreign Key	Step ID from the STEP_RESULT table that associates the result with a step.

Column Name	Data Type	Description
SEQUENCE_NAME	String	The name of the sequence that was called.
SEQUENCE_FILE_PATH	String	The path to the sequence file.

PROP_RESULT Table Schema

The default TestStand schema logs information about the properties in a step result to this table for steps. If necessary, the default TestStand schema logs additional information about specific property data types in the other tables named with the PROP prefix.

Table 16. PROP_RESULT Table Schema

Column Name	Data Type	Description
ID	Primary Key	Unique value that identifies each entry in the table.
STEP_RESULT	Foreign Key	Step ID from the STEP_RESULT table that associates the result with a step.
PROP_PARENT	Foreign Key	Property ID from the PROP_RESULT table that associates the result with a parent result.
ORDER_NUMBER	Number	The order in which the property results are processed for a step result.
NAME	String	Name of the property.
PATH	String	Descriptive string path to the property within the step result.
CATEGORY	Number	Type of property. Valid values include 0 (unknown), 1 (built-in), 2 (custom), 3 (additional), 4 (parameter).
TYPE_VALUE	Number	The property data type. Valid values include 0 (container), 1 (string), 2 (Boolean), 3 (number), and 5 (reference).
TYPE_NAME	String	Display name of the property data type.
DISPLAY_FORMAT	String	Numeric display format of the property.
DATA	String	Value of the property. The default schema sets this value for all intrinsic values, which includes string, numeric, Boolean, and Enumeration values.



Note The default TestStand schema logs the value of the `PropertyObjectType.DisplayString` property to the `TYPE_NAME` column of the `PROP_RESULT` table. Some type name values are localized in TestStand resource strings and will change if you localize TestStand resource strings. You can modify the **Value to Log** field in the `TYPE_NAME` column to `Logging.PropertyResultDetails.Type.TypeName` to log the type name.



Note The default TestStand schemas only log numeric values for Enumeration data types. Refer to [Logging Enumerations as Strings](#) for more information on how to log the enumerator strings in addition to their numeric values.

PROP_BINARY Table Schema

The default TestStand schema logs array data, such numeric arrays, to this table.

Table 17. PROP_BINARY Table Schema

Column Name	Data Type	Description
ID	Primary Key	Unique value that identifies each entry in the table.
PROP_RESULT	Foreign Key	Property ID from the PROP_RESULT table that associates the result with the generic property result.
UPPER_BOUNDS	String	Upper bounds of the data portion of the array. For example, "[4][5]".
LOWER_BOUNDS	String	Lower bounds of the data portion of the array. For example, "[0][0]".
DATA FORMAT	String	Data format of the DATA field. Refer to Logging Binary and String Database Values for more information about writing values to binary columns.
DATA	Binary	Values of the array.

PROP_ANALOGWAVEFORM Table Schema

The default TestStand schema logs analog waveform data, such as the

LabVIEWAnalogWaveform, NI_IviWave, and NI_IviWavePair data types, to this table.

Table 18. PROP_ANALOGWAVEFORM Table Schema

Column Name	Data Type	Description
ID	Primary Key	Unique value that identifies each entry in the table.
PROP_RESULT	Foreign Key	Property ID from the PROP_RESULT table that associates the result with the generic property result.
INITIAL_T	Date/Time	Trigger time of the waveform.
DELTA_T	Number	Time interval in seconds between data points in the waveform.
INITIAL_X	Number	Offset of the waveform.
DELTA_X	Number	Interval between data points in the waveform.
UPPER_BOUNDS	String	Upper bounds of the waveform. For example, "[4][5]".
LOWER_BOUNDS	String	Lower bounds of the waveform. For example, "[0][0]".
DATA_FORMAT	String	Data format of the DATA field. Refer to Logging Binary and String Database Values for more information about writing values to binary columns.
DATA	Binary	Values of the waveform.
ATTRIBUTES	String	Attributes information associated with the waveform.

PROP_DIGITALWAVEFORM Table Schema

The default TestStand schema logs analog waveform data, such as the LabVIEWDigitalWaveform data type, to this table.

Table 19. PROP_DIGITALWAVEFORM Table Schema

Column Name	Data Type	Description
ID	Primary Key	Unique value that identifies each entry in the table.
PROP_RESULT	Foreign Key	Property ID from the PROP_RESULT table that associates the result with the generic property result.

Column Name	Data Type	Description																		
INITIAL_T	Date/ Time	Trigger time of the waveform.																		
DELTA_T	Number	Time interval in seconds between data points in the waveform.																		
UPPER_BOUNDS	String	Upper bounds of the data portion of the waveform. For example, "[4][5]".																		
LOWER_BOUNDS	String	Lower bounds of the data portion of the waveform. For example, "[0][0]".																		
TRANSITIONS	Binary	1D array of 4-byte integers (I4) in which each element represents the transition location for the corresponding sample in data.																		
DATA	Binary	1D or 2D array of 1-byte integer (I1) values. Valid values include the following:																		
		<table><tr><td>Value</td><td>Digital Data State</td></tr><tr><td>0</td><td>0 (force down)</td></tr><tr><td>1</td><td>1 (force up)</td></tr><tr><td>2</td><td>Z (force off)</td></tr><tr><td>3</td><td>L (compare low)</td></tr><tr><td>4</td><td>H (compare high)</td></tr><tr><td>5</td><td>X (compare unknown)</td></tr><tr><td>6</td><td>T (compare off)</td></tr><tr><td>7</td><td>V (compare valid)</td></tr></table>	Value	Digital Data State	0	0 (force down)	1	1 (force up)	2	Z (force off)	3	L (compare low)	4	H (compare high)	5	X (compare unknown)	6	T (compare off)	7	V (compare valid)
		Value	Digital Data State																	
		0	0 (force down)																	
		1	1 (force up)																	
		2	Z (force off)																	
		3	L (compare low)																	
		4	H (compare high)																	
		5	X (compare unknown)																	
		6	T (compare off)																	
7	V (compare valid)																			
ATTRIBUTES	String	Attributes information associated with the waveform.																		

PROP_NUMERICLIMIT Table Schema

The default TestStand schema logs the limit information for a Numeric Limit Test step and a Multiple Numeric Limit Test step to this table.

Table 20. PROP_NUMERICLIMIT Table Schema

Column Name	Data Type	Description
ID	Primary Key	Unique value that identifies each entry in the table.
PROP_RESULT	Foreign Key	Property ID from the PROP_RESULT table that associates the result with the generic property result.
COMP_OPERATOR	String	The comparison operator for the numeric limit result.
THRESHOLD_TYPE	String	The type of threshold used by the numeric limit step type when using the EQT (== +/-) comparison type.
NOMINAL_VALUE	String	The base limit value of the numeric limit step, when you use the EQT (== +/-) comparison type.
UPPER_THRESHOLD	String	The higher threshold value, which is used along with the Nominal Value and the Threshold type to compute the HIGH_LIMIT.
LOWER_THRESHOLD	String	The lower threshold value, which is used along with the Nominal Value and the Threshold type to compute the LOW_LIMIT.
HIGH_LIMIT	String	The high limit for the numeric limit comparison.
LOW_LIMIT	String	The low limit for the numeric limit comparison.
UNITS	String	A label that describes the units for the measurement.
STATUS	String	The status of the numeric limit comparison.

Adding Support for Other Database Management Systems

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

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

Alternatively, you can create result tables for the default table schema for a similar DBMS by using the SQL script files located in the <TestStand>\Components\Models\

TestStandModels\Database directory and modifying the schema for the new DBMS.

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

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

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



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

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

```
DROP TABLE UUT_RESULT
/
```

```
DROP SEQUENCE SEQ_UUT_RESULT
/
DROP FUNCTION UUT_RESULT_NEXT_ID
/
```

Related concepts:

- [TestStand Directory Structure](#)
- [UUT_RESULT Table Schema](#)

Customizing Database Logging Expressions

When TestStand starts logging data to a database, it creates a temporary `Logging` property in the sequence context to evaluate expressions.

The `Logging` property contains subproperties that provide information about database settings, process model data structures, and the results TestStand processes. As the `Logging` property processes the result list, TestStand updates the subproperties of the `Logging` property to refer to the UUT result, step result, and the property result TestStand is processing. You can reference the `Logging` subproperties in the precondition and value expressions you specify for schema statements and columns.

The `Logging` property contains the following subproperties:

- `UUTResult`—The UUT result TestStand is processing. When TestStand is processing a step or a subproperty, this property contains the UUT result that contains the step result or subproperty.
- `StepResult`—The step result TestStand is processing. When TestStand is processing a subproperty, this property holds the step result that contains the subproperty. When TestStand is processing a UUT result, this property contains the result of the sequence call in the process model that calls the Main sequence in the client file.
- `PropertyResult`—The subproperty of the step result TestStand is processing. When TestStand is not processing a subproperty, this property does not exist.
- `PropertyResultDetails`—Information about the subproperty of the step result TestStand is processing. When TestStand is not processing a subproperty,

this container does not exist.

- `ExecutionOrder`—A numeric value TestStand increments after it processes each step result.
- `StartDate`—The date on which the UUT test began. This property is an instance of the `DateDetails` custom data type.
- `StartTime`—The time at which the UUT test began. This property is an instance of the `TimeDetails` custom data type.
- `UUT`—Specifies the serial number, test socket index, and other information about the UUT. This property is an instance of the `UUT` custom data type.
- `DatabaseOptions`—The process model database settings you configure in the Database Options dialog box. This property is an instance of the `DatabaseOptions` custom data type.
- `StationInfo`—The station ID and the user name. This property is an instance of the `StationInfo` custom data type.

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

Related concepts:

- [Sequence Context](#)

On-the-Fly Database Logging

When you enable the Use On-The-Fly Logging option on the Logging Options tab of the Database Options dialog box, the process models progressively log result data concurrently with the execution instead of waiting until UUT testing completes. The database logging plug-in uses the Model Plugin – OnTheFly Step Results entry point sequence to process the step results. The final data TestStand logs is almost identical to the data the process model generates at the end of execution.

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

When you use on-the-fly database logging with a schema that uses a stored procedure

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

The Generic Recordset (NI) and TS 2.x–4.0 Generic Recordset (NI) database schemas use the TOP 1 clause in the command text for all statements, as the following example shows, to improve performance when logging results to a database by preventing the recordset from accessing existing records in the database:

```
SELECT TOP 1 * FROM <TABLE NAME>
```

This performance improvement is useful when you enable the Use On-The-Fly Logging option for database files larger than 500 MB.

Related concepts:

- [Model Plugin – OnTheFly Step Results](#)
- [Default TestStand Table Schemas](#)

Logging Binary Strings in Databases

Typically, you cannot store binary data in a string since strings cannot contain zero bytes, and strings must not contain invalid multibyte character set (MBCS) sequences. However, the TestStand LabVIEW Adapter and the TestStand API function `SetValBinary` support assigning binary values to string variables or properties. TestStand stores binary string values as compressed and encoded binary data using only printable non-MBCS-lead-byte characters.

The default TestStand report generator process model plug-in generates the binary string values in the compressed encoded formats. See the Retrieving Binary String Values from Reports and Databases topic for more information about retrieving binary data from string values.

Related concepts:

- [Retrieving Binary String Values from Reports and Databases](#)

Database Known Issues

Consider the following issues when you use a database management system with TestStand:

- On 64-bit Windows operating systems, the Data Sources (ODBC) application in the Windows Control Panel is the 64-bit version of the application, located at `C:\Windows\system32\odbcad32.exe`. NI recommends that you use the 32-bit version of the ODBC application, located at `C:\Windows\SysWOW64\odbcad32.exe` to create any system DSNs for use in 32-bit TestStand. If you create a system DSN using the 64-bit version, the ODBC application creates a 64-bit DSN. If you create a system DSN using the 32-bit version, the ODBC application creates a 32-bit DSN. If you attempt to use system data links from the 32-bit TestStand Database Options dialog box or the Database Viewer, TestStand launches the 32-bit version of the Data Link Properties dialog box, which cannot display 64-bit system DSNs. User defined DSNs can be configured from the 32-bit version or the 64-bit version of the Data Sources (ODBC) application. If you attempt to use data links from the Database Options dialog box or the Database Viewer, the Data Link Properties dialog box matching TestStand's bitness will be launched. A 32-bit or 64-bit version of this dialog can only display 32-bit or 64-bit DSNs respectively. Therefore, ensure that the DSNs are configured in the correct version of Data Sources (ODBC) application for it to be visible from TestStand

▼ Microsoft Access Issues

- You must enable the **Share Data Link Between Executions** option on the Database Options dialog box to eliminate possible contention for access to a shared database file from multiple executions, and when you enable more than one database result processing plug-ins in the Result Processing dialog box.
- TestStand 3.5 or later includes a workaround for incorrect database logging when using the Microsoft Jet Provider 4.0. The Microsoft Jet Provider 3.51 and 4.0 incorrectly swap the month and day numeric components under the following conditions:
 - The Regional Options on the Microsoft Windows Control Panel specify a locale with a default date format that shows the day before the month

- The TestStand Database Options data link uses the Microsoft Jet 3.51 or 4.0 Provider
- The TestStand Database Options schema uses parameterized INSERT statements and includes a column defined with a Date data type
- The day in the date falls between the 1st and the 12th of the month.

For example, when TestStand logs the value January 13, 2012, the Microsoft Jet Provider stored date is correct. However, when TestStand logs the value January 12, 2012, the Microsoft Jet Provider stored date is December 1, 2012.

- The Database Viewer application fails to view data inside a Microsoft Access database when the data link uses the Microsoft Open Database Connectivity (ODBC) Driver and the base filename contains a period character in addition to the period in the file extension. This error appears to be a problem with the Microsoft ODBC Driver. This error may make the Database Viewer application unstable. Rename the database file or use the Microsoft Jet 4.0 Object Linking and Embedding (OLE) DB Provider instead.
- When the Microsoft Access ODBC Driver and Jet Provider return the error `Operation cancelled`, verify that the ODBC data source or data link connection string references a valid `.mdb` file.
- The Microsoft Jet 4.0 OLE DB Provider and ODBC Driver might return an error message when more than one execution or thread attempts to gain write access to the same table because executions and threads do not share handles when opening unique connections to the database or executing unique statements on a table. The Access drivers use an on-demand file access method that can cause this error.

Take one of the following actions to resolve this issue:

- When this error occurs while using the TestStand database logging feature from within a single process, enable the **Share Data Link Between Executions** option on the Data Link tab of the Database Options dialog box.
- When this error occurs while accessing an Access database from multiple processes or different computers, ignore the error or retry the failing database action.
- In some cases, when you open the statement recordset using the Batch Optimistic Locking option, the problem might not occur.
- When you use the Microsoft Access ODBC Driver, values for the STEP_TYPE field in

the STEP_RESULT table might be NULL because MDAC sets the values for a MEMO field to NULL for all records when the value of the first record written is not specified. By default, the script files used to create the table attempt to create TEXT fields. When the tables are created with the Access ODBC driver, the field type is TEXT of size 255. When the tables are created with the Access OLE DB Provider, the field type is MEMO.

Complete one of the following actions to resolve this issue:

- Use the Microsoft Jet 4.0 OLE DB Provider instead of the Access ODBC driver to log results.
- Change the field types in the STEP_RESULT and UUT_RESULT tables from MEMO to TEXT within Access. A message warns you that data could be truncated to 255 characters, which is the limit of a TEXT field.

▼ MySQL Issues

- The MySQL ODBC drivers incorrectly report a catastrophic failure error when you specify a connection string that includes the "**Initial Catalog=xxx**" parameter. Specify the default database in the ODBC data source using the ODBC Administrator to resolve this issue.
- When you use the MySQL ODBC driver and the operating computer specifies a comma character as the decimal symbol character, the ODBC driver might return an error because the ODBC driver internally converts a floating-point value to a string value. The computer locale causes MySQL to interpret the comma decimal symbol character in the SQL syntax as a multi-value list character separator. Configure the MySQL data source in the ODBC Administrator and enable the **Don't Use Set Locale** option in the Miscellaneous Options section to resolve this error.
- The MySQL ODBC driver returns an error while logging results to a database when you specify column data type as `String (BSTR)` because the MySQL ODBC driver fails to encapsulate string values in quotation marks when constructing SQL statements. Specify string columns as `String (varchar)` instead.
- The MySQL ODBC driver returns an error while logging results to the database when the schema in the Database Options dialog box specifies a column data type as `GUID`. Specify string columns as `String (varchar)` instead.
- The MySQL ODBC 3.51 driver ignores the initial catalog setting specified in this dialog box or the Database Viewer application. You must specify the database in the ODBC connection string or in the DSN specified in the ODBC Administrator.

- The MySQL ODBC 3.51 driver might return an error when the Database Viewer application attempts to view a different database when you change the name in the Catalog ring control. The MySQL ODBC driver internally fails to properly terminate the string that contains the database name, which causes extra characters to appear on the end of the database name.
- The MySQL ODBC 3.51 driver does not permit the creation of new empty records when using a SELECT statement. When you attempt to use the On-The-Fly Database Logging option with schemas that use SELECT statements, the MySQL database returns the following error: Empty row cannot be inserted. Row must have at least one column value set.
- Enable the **Don't Optimize Column Width** option when you configure the DSN in the ODBC Administrator dialog box to prevent the MySQL ODBC 3.51 driver from returning the Operation is not allowed in this context (-2146825069) error when logging data to a binary field.
- When you perform a validate operation, the MySQL ODBC 3.51 driver incorrectly returns a maximum length of 64,000 for LONGBLOB or MEDIUMBLOB columns.
- NI recommends not using the Fetch Record by Index operation mode of the Data Operation database step type when reading a table with a VARCHAR field. In cases where multiple VARCHAR fields exist in the database and the Fetch Record By Index operation executes multiple times, a System Level Exception might occur. Instead, use the Next – Fetch Next Record operation to step through records in the table or use the Fetch Record by Index operation if the offset from the current record to the record index is small.
- A MySQL connection string requires that you specify the Initial or Default schema to which you are connecting. If this field is not specified or left empty in the connection string, Memory Access Violation errors occur when you connect to MySQL using the Database Viewer the MySQL ODBC Driver version 5.2 or later, for example, `Driver={MySQL ODBC 5.2 ANSI Driver};Server=servname;Database=TSInsert;User=tester;Password=s`
- MySQL returns Access Violation errors when retrieving metadata on the database schema if there are no relevant objects to retrieve. For example, if the application attempts to fetch the list of views in a MySQL schema and no views are defined, an Access Violation error occurs. You can ignore this error with no negative consequences.
- MySQL returns a MySQL server has gone away error when when using MySQL for database logging, the On-The-Fly Reporting option is enabled, and you run the sequence using the batch or parallel model.

▼ SQL Server Issues

- Memory usage can increase when using the SQL Server Stored Proc schema and the database connection string uses the Microsoft OLE DB Provider for SQL Server. The memory usage appears to oscillate but shows growth over time. Use the Microsoft SQL Native Client database provider to avoid this issue.

▼ Oracle Issues

- Use MDAC 2.7 with Oracle to prevent error -2147217915, which occurs when you execute the stored procedure for the `STEP_RESULT` schema statement.
- When you use an Oracle stored procedure to log data to a foreign key column that uses the GUID type and the data type of the parameter in the database logging schema is GUID, Oracle returns error -2147467259 when no value or `NULL` is assigned to the parameter. Change the data type from GUID to String for the foreign key parameter in the schema to resolve this issue.
- The Database Viewer application cannot show the contents of an Oracle database table when the table contains a LONG column. The Microsoft data grid control returns an "Object is open" error.
- Oracle recommends using Oracle Provider 11.1.0.6.0 or later if you want to read 64-bit integer values stored in NUMBER columns.
- When you enable the **Share Data Link Between Execution** option and multiple executions simultaneously log binary data to a BLOB column in an Oracle database, Oracle can return the error "ORA-22990: LOB locators cannot span transactions." Disable the **Share Data Link Between Execution** option or enable the **Use Transaction Processing** option to prevent this error.
- When you use the Oracle ODBC Driver with server-side cursors to log data in TestStand with On-The-Fly logging enabled, the Oracle ODBC driver fails to return a proper key value for a parent provisional result of a Sequence Call step. Consequently, TestStand cannot update the fields of the parent provisional result and instead creates a new Sequence Call step result, and the foreign keys values for the steps in the called sequence incorrectly reference the empty provisional result. To work around this limitation of the Oracle driver, configure statements to use client-side cursors for the schema in the Statements tab of the Database Options dialog box. NI recommends using the Oracle OleDb provider with server-side cursors to avoid this issue with the Oracle ODBC Driver.

▼ Sybase Issues

- The Sybase Adaptive Server Anywhere ODBC and OLE DB Provider do not store string data properly into a column when logging results to database when you specify the column data type as `String (BSTR)`. Specify string columns as `String (varchar)` instead.

Related concepts:

- [STEP_RESULT Table Schema](#)
- [UUT_RESULT Table Schema](#)
- [On-the-Fly Database Logging](#)
- [ODBC Data Source Administrator Dialog Box](#)

Logging Enumerations as Strings

The default TestStand schemas only log the numeric values for Enumeration data types in the `PROP_RESULT` and `PROP_BINARY` tables.

To log the name/string value of all Enumeration types, you can modify the schema for the `PROP_RESULT` table in the following manner:

1. Open **Configure»Result Processing»Database** to edit the schema TestStand uses for Database logging.
2. Select the **DATA** column in the `PROP_RESULT` table of the schema you are using and modify the Value To Log field to use a conditional expression:

```
Logging.PropertyResultDetails.Type.ValueType==7?  
Str(Logging.PropertyResult):Logging.PropertyResult
```



Note If you are using a default schema, make a copy of the schema and edit the copy to make these changes.

Similarly, you can use the `GetValString` method on the `Logging.PropertyResult` object to obtain the string value of an enum and log it in a new column.



Note The new expression in Value To Log will give an evaluation error at edit-time, but evaluates correctly at run-time.

To Log the string **and** numeric value of all enumeration types, modify the expression in step 2 above to:

```
Logging.PropertyResultDetails.Type.ValueType==7?
Logging.PropertyResult.GetValueDisplayName("",0):Logging.PropertyResult
```

To log an array of enumeration as a single string, modify the schema for the PROP_RESULT table as follows:

1. Open **Configure»Result Processing»Database** to edit the schema TestStand uses for Database logging.
2. Select the **DATA** column in the PROP_RESULT table of the schema you are using. Modify the Precondition expression from

```
(Logging.PropertyResultDetails.Type.ValueType>0&&
Logging.PropertyResultDetails.Type.ValueType<4) ||
Logging.PropertyResultDetails.Type.ValueType==7
```

to

```
(Logging.PropertyResultDetails.Type.ValueType<0&&
Logging.PropertyResultDetails.Type.ValueType<4) ||
Logging.PropertyResultDetails.Type.ValueType==7 ||
(Logging.PropertyResultDetails.Type.ValueType== 6/*array*/&&
Logging.PropertyResultDetails.Type.ElementType.ValueType==7/*enum*/)
```

3. Modify the Value To Log expression to

```
(Logging.PropertyResultDetails.Type.ValueType==6/*array*/&&Logging.
PropertyResultDetails.Type.ElementType.ValueType==7/*enum*/)?
Str(Logging.PropertyResult,"%s",1,False,"."):Logging.PropertyResult
```



Note This logs the enumerator array as a single string composed of the string values of the enumerator array elements delimited by a comma. The data will be logged to the PROP_RESULT table.

Related concepts:

- [PROP_RESULT Table Schema](#)
- [PROP_BINARY Table Schema](#)

Logging Attribute Values

The default TestStand database schemas do not log attribute values that exist on result properties. The following examples demonstrate how you can customize a database schema in the Database Options dialog box to log attributes.

Logging a Specific Attribute to a Single Field

To log the value of a specifically-named attribute that can exist on a specific result property, retrieve the attribute value and store the value in a new field in a table.

For example, if the code modules stored optional, additional reason information in the attribute `MyNamespace.StatusReason` on the `Status` property of steps, you can log the value of this attribute to a new field in the `STEP_RESULT` table by creating a new String column in the `STEP_RESULT` table and configuring the schema to include the following new column settings:

Column Name	STATUS_REASON
Type	String
Size	255
Direction	Input (To DB)
Expected Properties	Logging.StepResult.Status.Attributes.MyNamespace.StatusReason
Precondition	–
Expression	Logging.StepResult.Status.Attributes.MyNamespace.StatusReason

Logging Attributes for Properties

To log all attribute names and values for any property logged to the `PROP_RESULT` table, which includes most custom step properties and additional results, serialize the parent `Attributes` property for each result property and store the serialization to a new field in the `PROP_RESULT` table. For example, you can add a new string column, `ATTRIBUTES`, to the `PROP_RESULT` table and configure the schema to include the following new column settings:

Column Name	ATTRIBUTES
Type	String
Size	4096
Direction	Input (To DB)
Expected Properties	–
Precondition	Logging.PropertyResult.HasAttributes
Expression	Logging.PropertyResult.Attributes.GetXML(0xF0F, 0)



Note A client application that retrieves serialized attribute information from the database might be required to unserialize to process the attribute information.

Generating and Customizing TestStand Reports

Consult the topics in this section to learn how to generate a report for a test execution and how to customize the content and format of the report.

In addition, this section describes how to choose the appropriate report generation strategy for a test system by reviewing how the report generation features in TestStand can help meet requirements and by describing the tradeoffs to consider when designing report generation for a test system. Use cases illustrate how common test system characteristics and requirements map to specific report generation strategies.

Related concepts:

- [Choosing the Appropriate Report Generation Strategy](#)

Choosing the Appropriate Report Generation Strategy

Most automated test systems need to generate a report that logs information about each test run against the unit under test (UUT), including the test status, measurement results, test parameters, and diagnostic information collected during the test. Test systems also have system-level requirements and constraints—such as hardware

platform, test throughput, tester up-time, and linkage to Manufacturing Execution Systems (MES) or other enterprise management systems—that can influence the report generation strategy the test system uses.

In most cases, the report generation strategy defined for one test system might not meet the requirements of another test system, even within the same organization. Test system architects must typically consider many or all of the following requirements when defining the appropriate report generation strategy for a particular test system:

- Maximize test system throughput
- Minimize report file size
- Include all necessary report content
- Interoperate with other processes and systems
- Support post-failure information recovery
- Generate reports immediately for the current UUT before testing the next UUT
- Generate multiple reports for each UUT
- Generate and view the report during test sequence execution

The topics in this section examine each of these requirements, recommend TestStand report generation settings and configurations that best meet each requirement in typical scenarios, and identify other requirements you might have to sacrifice to optimize a test system for the requirement. NI recommends that you use the information in this section as a starting point for designing a report generation strategy for a test system and iteratively measure and modify the test system to achieve the optimal balance among the requirements.

Consider the different scenarios in which you run a test system and which requirements are most important for each scenario. For example, when you first develop and debug a test system, throughput might not matter, but support for post-failure information recovery might be critical. When you move to production, throughput might be the most important requirement.



Note

- In some cases, a test system might specify a unique set of requirements that the built-in report generators cannot satisfy. NI recommends that

you choose a report format that most closely satisfies the set of requirements and modify it to fit the requirements. You can create custom result processing model plug-ins to generate such a report. For example, a test system might require the performance throughput of an ASCII report format, the interoperability of an XML report format, and the ability to log a small amount of data to the report. You can create a custom result processing model plug-in that generates an XML report that contains only the information you require.

- You can use multiple results processing configurations to switch easily between settings based on test system execution scenarios.

Related concepts:

- [Maximizing Test System Throughput](#)
- [Minimizing Report File Size](#)
- [Including All Necessary Report Content](#)
- [Interoperating with Other Processes and Systems](#)
- [Supporting Post-Failure Information Recovery](#)
- [Generating the Report for the Current UUT before Testing the Next UUT](#)
- [Generating Multiple Reports for Each UUT](#)
- [Generating and Viewing the Report during Test Sequence Execution](#)
- [Process Model Plug-In Architecture](#)

Maximizing Test System Throughput

Automated test systems are usually designed to maximize throughput by minimizing the time required to verify a unit under test (UUT). Test system architects spend a considerable amount of time optimizing test system performance. In a typical test system, the amount of time required for report generation depends on a number of factors, including available resources (processor and memory), the format of the report to generate, the number of results, and additional data to log.

Recommended Report Options

NI benchmarks demonstrate the following patterns based on the report format, the

number of step results or the size of the data sets you log, the number of cores in the test computer, and other options you configure in the Report Options dialog box:

- When you log 500 or fewer step results on a single- or multi-core computer, ASCII reports have the least negative impact to throughput, and XML reports have the most negative impact to throughput.
- When you log more than 500 step results or large data sets on a single- or multi-core computer, TSR reports have the least negative impact to throughput, and XML reports have the most negative impact to throughput.
- Asynchronous report generation on a single-core computer does not improve throughput, except in cases where a substantial amount of sequence execution time is spent waiting on operations that do not use the CPU, such as instrument I/O.
- Asynchronous report generation on a multi-core computer improves throughput for all report formats.

The following table summarizes the recommended options you configure in the Report Options dialog box, depending on your system configuration, when you optimize for test system throughput:

Report Option	Single-Core Computers		Multi-Core Computers	
	Value for ≤ 500 Step Results	Value for ≥ 500 Step Results	Value for ≤ 500 Step Results	Value for ≥ 500 Step Results
Format	ASCII, TSR*, ATML, HTML, XML (best to worst)	TSR, ASCII, ATML, HTML, XML (best to worst)	ASCII, ATML, TSR, HTML, XML (best to worst)	TSR, ASCII, ATML, HTML, XML (best to worst)
Asynchronous	False	False	True	True
On-the-Fly Report	False	False	False	False
Only Display Latest Results	False	False	False	False



Note *The TSR format generates a compact raw results file and, if used, requires the TestStand Offline Results Processing Utility to generate the final report.

Tradeoffs

Consider the impact to the following requirements when you optimize for test system throughput:

- Minimize report file size
- Include all necessary report content
- Interoperate with other processes and systems
- Support post-failure information recovery
- Generate reports immediately for the current UUT before testing the next UUT
- Generate multiple reports for each UUT
- Generate and view the report during test sequence execution

Related concepts:

- [Minimizing Report File Size](#)
- [Including All Necessary Report Content](#)
- [Interoperating with Other Processes and Systems](#)
- [Supporting Post-Failure Information Recovery](#)
- [Generating the Report for the Current UUT before Testing the Next UUT](#)
- [Generating Multiple Reports for Each UUT](#)
- [Generating and Viewing the Report during Test Sequence Execution](#)
- [Choosing the Appropriate Report Generation Strategy](#)

Minimizing Report File Size

Automated test systems can generate a significant amount of data when testing a unit under test (UUT). The amount of data generated depends on the complexity of the UUT to test and the capabilities of the test equipment. The size of the report generated depends on the format of the report and the amount of data to log. Primarily, system-level requirements of storage for traceability or analysis and the testing and debugging processes you use influence the amount of data to log in the report. In all cases, NI benchmarks show that the TSR report format is the most efficient at minimizing size on disk.

Recommended Report Options

The following table summarizes the recommended options you configure in the

Report Options dialog box when you optimize for report size:

Report Option	Value
Format	TSR*, ASCII, ATML, HTML, XML (best to worst)
Asynchronous	N/A
On-the-Fly Report	N/A
Only Display Latest Results	N/A



Note *The TSR format generates a compact raw results file and, if used, requires the TestStand Offline Results Processing Utility to generate the final report.

Tradeoffs

Consider the impact to the following requirements when you optimize for report size:

- Maximize test system throughput
- Include all necessary report content
- Interoperate with other processes and systems

Related concepts:

- [Maximizing Test System Throughput](#)
- [Including All Necessary Report Content](#)
- [Interoperating with Other Processes and Systems](#)

Including All Necessary Report Content

Test systems have different data logging requirements influenced by system-level requirements of storage for traceability or analysis and by the testing and debugging processes used.

Each report format TestStand can generate contains different pieces of information. For example, the ASCII and HTML report formats contain only the step name, status, limits, limit comparison type, module time, and any additional results associated with

the step. The TSR format contains all the information available in the ASCII and HTML formats and additional information, such as the parent step IDs of steps and the execution thread IDs.

Use the following categories to consider the set of data available in each report format:

- **Minimal**—Contains minimal context information, such as the station ID, user name, execution start date and time, overall status, and minimal step-related information (such as the step name, module time, measurement value, and limits and module time).
- **Controlled**—Contains all the information in a minimal report and information associated with a step, such as step ID.
- **Moderate**—Contains all the information in a controlled report and other information, such as the step type, step group, total step execution time, and report options.
- **Complete**—Contains all the information in a moderate report and other information, such as the parent step IDs and execution thread IDs. The `ResultLogRecordTypes` enumeration specifies the complete list of all data logged.

The following table summarizes how the built-in TestStand report formats correspond to these report type categories:

Report Type	ASCII	ATML 6.01, 5.0, or 2.02	ATML 6.01, 5.0, or 2.02 with Extensions	HTML	TSR	XML
Minimal	✓	—	—	✓	—	—
Controlled	—	✓	—	—	—	—
Moderate	—	—	✓	—	—	✓
Complete	—	—	—	—	✓	—

Recommended Report Options

The following table summarizes the recommended options you configure in the Report Options dialog box when you optimize for report content:

Report Option	Value
Format	ATML, XML, TSR*, ASCII, HTML (best to worst)
Asynchronous	N/A
On-the-Fly Report	N/A
Only Display Latest Results	N/A



Note *The TSR format generates a compact raw results file and, if used, requires the TestStand Offline Results Processing Utility to generate the final report.

Tradeoffs

Consider the impact to the following requirements when you optimize for report content:

- Maximize test system throughput
- Minimize report file size
- Interoperate with other processes and systems

Related concepts:

- [Maximizing Test System Throughput](#)
- [Minimizing Report File Size](#)
- [Interoperating with Other Processes and Systems](#)
- [Customizing the ATML Test Results Report Generator](#)

Interoperating with Other Processes and Systems

Many test systems integrate with an Enterprise Resource Planning (ERP) system or a Manufacturing Execution System (MES) that processes test system reports to provide business-critical data to other systems. In some cases, an engineer or technician might process the report for statistical analysis on a separate computer. These post-processing requirements might dictate the report generation format for the test system.

TestStand saves the ATML and XML reports in structured XML file formats that are easily processed using an API for XML (SAX) or DOM parsers. TestStand saves the ASCII and HTML reports in unstructured ASCII file formats that are not as easy to post-process as XML file formats. NI defines the TSR file format, which requires the TestStand Engine and the TestStand Offline Results Processing Utility to generate another file in a format you can process directly.

Recommended Report Options

The following table summarizes the recommended options you configure in the Report Options dialog box when you optimize for interoperating with other processes and systems:

Report Option	Value
Format	ATML, XML, ASCII, HTML, TSR* (best to worst)
Asynchronous	N/A
On-the-Fly Report	N/A
Only Display Latest Results	N/A



Note *The TSR format generates a compact raw results file and, if used, requires the TestStand Offline Results Processing Utility to generate the final report.

Tradeoffs

Consider the impact to the following requirements when you optimize for interoperating with other processes and systems:

- Maximize test system throughput
- Minimize report file size
- Include all necessary report content
- Support post-failure information recovery
- Generate reports immediately for the current UUT before testing the next one
- Generate multiple reports for each UUT

Related concepts:

- [Maximizing Test System Throughput](#)
- [Minimizing Report File Size](#)
- [Including All Necessary Report Content](#)
- [Supporting Post-Failure Information Recovery](#)
- [Generating the Report for the Current UUT before Testing the Next UUT](#)
- [Generating Multiple Reports for Each UUT](#)

Supporting Post-Failure Information Recovery

In the case of a catastrophic failure, analyzing step result information collected during the execution might be a critical component of the troubleshooting process.

You can configure TestStand to generate reports on-the-fly while testing a unit under test (UUT). By default, TestStand generates and persists the on-the-fly report after collecting 500 step results or after every 1.5 seconds of execution time, whichever occurs first. Logging reports to disk can be slow and negatively impact throughput.

In some cases, you can improve performance by changing the values of the **Processing Interval** or the **Maximum Number of Results** options in the Advanced Result Processing Settings dialog box. You can also improve performance by enabling the **Only Display Latest Results** option on the Contents tab of the Report Options dialog box.

NI benchmarks show that TSR reports provide better throughput while generating an on-the-fly report.

Recommended Report Options

The following table summarizes the recommended options you configure in the Report Options dialog box when you optimize for post-failure information recovery:

Table 21. Recommended Report Options

Report Option	Value
Format	TSR*, ATML, XML, ASCII, HTML (best to worst)
Asynchronous	N/A

Report Option	Value
On-the-Fly Report	True
Only Display Latest Results	True



Note *The TSR format generates a compact raw results file and, if used, requires the TestStand Offline Results Processing Utility to generate the final report.

Tradeoffs

Consider the impact to the following requirements when you optimize for post-failure information recovery:

- Maximize test system throughput
- Interoperate with other processes and systems
- Generate and view the report during test sequence execution

Related concepts:

- [Maximizing Test System Throughput](#)
- [Interoperating with Other Processes and Systems](#)
- [Generating and Viewing the Report during Test Sequence Execution](#)
- [Generating the Report for the Current UUT before Testing the Next UUT](#)
- [On-the-Fly Report Generation](#)

Generating the Report for the Current UUT before Testing the Next UUT

At the end of the test sequence execution, some systems require moving a printout of a report associated with the unit under test (UUT) to the next stage of the manufacturing process along with the UUT. For such reports, raw results generators and asynchronous report generation are not appropriate. This requirement does not impact other report formats.

You can generate on-the-fly reports to satisfy this requirement because TestStand generates on-the-fly reports synchronously.

Recommended Report Options

The following table summarizes the recommended options you configure in the Report Options dialog box when you optimize for ensuring the report for a UUT is generated before the next UUT is tested:

Report Option	Value
Format	ASCII, ATML, XML, HTML (best to worst)
Asynchronous	False
On-the-Fly Report	True
Only Display Latest Results	N/A

Tradeoffs

Consider the impact to the following requirements when you optimize for ensuring the report for a UUT is generated before the next UUT is tested:

- Maximize test system throughput
- Generate multiple reports for each UUT
- Generate and view the report during test sequence execution

Related concepts:

- [Maximizing Test System Throughput](#)
- [Generating Multiple Reports for Each UUT](#)
- [Generating and Viewing the Report during Test Sequence Execution](#)
- [Supporting Post-Failure Information Recovery](#)
- [On-the-Fly Report Generation](#)

Generating Multiple Reports for Each UUT

Some test systems might specify a requirement to generate more than one report for each unit under test (UUT). For example, you might want to generate a fast and simple summary ASCII report and a detailed ATML report that can be post-processed by a Manufacturing Execution System (MES). You might also want to generate a summary report for customers and a detailed report for internal use.

You can configure TestStand to generate multiple reports for each UUT. NI benchmarks show that generating a raw results file and processing it offline to generate multiple reports yields the maximum throughput on multi-core computers.

Recommended Report Options

The following table summarizes the recommended options you configure in the Report Options dialog box when you optimize for generating multiple reports:

Report Option	Value
Format	TSR*, ASCII, ATML, XML, HTML (best to worst)
Asynchronous	True
On-the-Fly Report	False
Only Display Latest Results	False



Note *The TSR format generates a compact raw results file and, if used, requires the TestStand Offline Results Processing Utility to generate the final report.

Tradeoffs

Consider the impact to the following requirements when you optimize for generating multiple reports:

- Maximize test system throughput
- Interoperate with other processes and systems
- Generate reports immediately for the current UUT before testing the next one
- Generate and view the report during test sequence execution

Related concepts:

- [Maximizing Test System Throughput](#)
- [Interoperating with Other Processes and Systems](#)
- [Generating the Report for the Current UUT before Testing the Next UUT](#)
- [Generating and Viewing the Report during Test Sequence Execution](#)

Generating and Viewing the Report during Test Sequence Execution

Some test procedures can require several hours or more to complete because the procedures contain a significant number of tests or contain a small number of tests that take a long time to complete. To help diagnose issues early in such test systems, you can generate and display a partial report while the test sequence executes instead of waiting until the end of execution to generate and display the report.

You can configure TestStand to generate a report on-the-fly to support such test sequences. NI benchmarks show that if you want to view the entire report as the testing progresses, an on-the-fly ASCII report provides the best throughput. If you do not need to see the entire report, all reports generated on-the-fly when you enable the **Only Display Latest Results** option on the Contents tab of the Report Options dialog box provide approximately equivalent throughput. Because you cannot generate the report associated with raw results until after the execution completes, you cannot use the raw results file format to generate and view the report during the test sequence execution.

Test sequences that specify a large number of steps generate very large reports. Generating and viewing a large report while executing a test sequence can cause significant performance issues. Therefore, when you generate reports on-the-fly, NI recommends that you enable the **Only Display Latest Results** option.

Recommended Report Options

The following table summarizes the recommended options you configure in the Report Options dialog box when you optimize for generating and viewing the report during test sequence execution:

Report Option	Value for a Sequence with Few Steps and Long Execution Time	Value for a Sequence with Many Steps
Format	ASCII, ATML, HTML, XML (best to worst)	ASCII, ATML, HTML, XML (best to worst)
Asynchronous	N/A	N/A
On-the-Fly Report	True	True
Only Display Latest	False	True

Report Option	Value for a Sequence with Few Steps and Long Execution Time	Value for a Sequence with Many Steps
Results		

Tradeoffs

Consider the impact to the following requirements when you optimize for generating and viewing the report during test sequence execution:

- Maximize test system throughput
- Interoperate with other processes and systems
- Support post-failure information recovery
- Generate reports immediately for the current UUT before testing the next UUT

Related concepts:

- [Maximizing Test System Throughput](#)
- [Interoperating with Other Processes and Systems](#)
- [Supporting Post-Failure Information Recovery](#)
- [Generating the Report for the Current UUT before Testing the Next UUT](#)
- [Generating Multiple Reports for Each UUT](#)
- [Use Case: Generate Different Types of Reports from the Same Data Set](#)
- [On-the-Fly Report Generation](#)

Use Case: Generate Different Types of Reports from the Same Data Set

Scenario

Consider a test system in which the report generation functionality must meet the following criteria:

- Maximize test system throughput
- Generate and display a summary report immediately after testing the unit under test (UUT)
- Generate a detailed report a Manufacturing Execution System (MES) can post-process

Solution

Create one results processing configuration to generate a high-level summary report and different configurations for more detailed reports.

Detailed Solution

For the summary report, create a result processing configuration that specifies the following options in the Report Options dialog box:

Report Option	Value
Format	ASCII
Asynchronous	False
On-the-Fly Report	False
Only Display Latest Results	False

For a more detailed report, create a result processing configuration that specifies the following options in the Report Options dialog box:

Report Option	Value
Format	ATML
Asynchronous	True
On-the-Fly Report	False
Only Display Latest Results	False

In some cases, the Manufacturing Execution System (MES) might not post-process the detailed report at the time the MES tests the UUT but instead at a later time or on a different computer. In this case, create a result processing configuration that specifies the following options in the Report Options dialog box so you can generate a TSR file and post-process results using the TestStand Offline Result Processing Utility, which can generate an ATML report the MES can post-process:

Report Option	Value
Format	TSR
Asynchronous	True
On-the-Fly Report	False
Only Display Latest Results	False

Related concepts:

- [Process Model Plug-In Architecture](#)
- [Choosing the Appropriate Report Generation Strategy](#)
- [Use Case: Log Data at One Location but Generate the Report at a Separate Location](#)

Use Case: Log Data at One Location but Generate the Report at a Separate Location

Scenario

Consider a test system that runs production tests off-site at a contract manufacturer facility. The report generation functionality must meet the following criteria:

- Support post-failure information recovery to help diagnose and isolate faults
- Store the report for each unit under test (UUT) for traceability or analysis

Solution

Generate a compact raw results file at the off-site test system location, transfer the TSR file on-site, and generate the final report using the TestStand Offline Result Processing Utility.

Detailed Solution

Create a result processing configuration that specifies the following options in the Report Options dialog box:

Report Option	Value
Format	TSR
Asynchronous	True
On-the-Fly Report	True
Only Display Latest Results	True

Related concepts:

- [Process Model Plug-In Architecture](#)
- [Use Case: Generate Different Types of Reports from the Same Data Set](#)

Generating a Test Report

You can generate a report to process collected test results. Most of the report generation capabilities are not native to the TestStand Engine or to the TestStand Sequence Editor. The default process model, which calls the Main sequence in the client sequence file to test a UUT, relies on the automatic result collection capabilities of the TestStand Engine to accumulate the raw data for each report. The engine automatically compiles the result of each step into a result list for an entire sequence, which contains the result of each step and the result list of each subsequence call made.

When you run a sequence using the Test UUTs or Single Pass Execution entry point, the default report generation process model plug-in, NI_ReportGenerator.seq, located in the <TestStand>\Components\Models\ModelPlugins directory, generates the report by traversing the results for the Main sequence in the client sequence file and all the subsequences called.

Configuring Report Options

Select **Configure»Result Processing** to launch the Result Processing dialog box. Place a checkmark in the Enabled column of the **Report** row to enable the default report generation process model plug-in. You can customize or replace any portion of the report generator process model plug-in.

Click the  icon in the Options column for the reporting model plug-in to launch the

Report Options dialog box, in which you can specify options that determine the content and format of the report and the names and locations of report files.

Viewing and Using Reports

You can use reports in the following ways in TestStand:

- The Report pane of the Execution window displays the report for the current execution. Usually, the Report pane is empty until execution completes.

Alternatively, you can use an external application to view reports. Select **Configure»External Viewers** to launch the Configure External Viewers dialog box, in which you can specify the external application TestStand launches to display a particular report format. In the Execution window, click the **Viewer** button on the Report pane to view the report in the external viewer you specified.

- You can generate reports on-the-fly to progressively log result data concurrently with the execution instead of waiting until UUT testing completes.
- You can generate reports in different formats. The default process model generates reports in Automatic Test Markup Language (ATML), XML, HTML, or ASCII-text formats.
- You can include hyperlinks in supported versions of ATML, XML, and HTML reports.
- You can distribute these types of report files to view on a computer that does not have TestStand installed.
- You can create a custom result processing model plug-in, and you can configure multiple instances of reporting model plug-ins in the Result Processing dialog box, in which case you can use the **Active Report** control on the Report pane toolbar to select which report to view.
- The report generator can generate a PDF report. Refer to Generating PDF Reports.



Note All TestStand report generators support generating reports that contain 64-bit integer data types. Signed 64-bit integers use the %d format string, and unsigned 64-bit integers use the %u format string.

Customizing Reports with Expressions

In addition to using options to determine the names and locations of report files, you

can specify a custom expression with predefined macros TestStand evaluates at run time for each UUT to fully customize the filename of reports and the directory in which TestStand stores reports. For example, you can use expressions to save reports in a directory that uses the sequence filename, save reports in different directories based on execution status, and generate report filenames that include the serial number, user name, or execution status.

You can use expressions to replicate all existing report filename options. Use the Report File Pathname tab of the Report Options dialog box to specify an expression for a report.

The following table lists common report options and the equivalent expression for the Sequential process model using the client sequence file directory as the base directory.

Current Report Option	Equivalent Expression
New UUT Report File for Each UUT	<code>\$(ClientFileDir)\\Report_\$(UUT).\$(FileExtension)</code>
Prefix Sequence File Name to Report File Name	<code>\$(ClientFileDir)\\\$(ClientFileName)_Report.\$(FileExtension)</code>
Add Date and Time to File Name	<code>\$(ClientFileDir)\\Report_\$(FileDate)\$ (FileTime).\$(FileExtension)</code>
Force File Name to Be Unique	<code>\$(ClientFileDir)\\Report_\$(Unique).\$(FileExtension)</code>
Prefix	<code>\$(ClientFileDir)\\\$(ClientFileName)_Report_\$(UUT)_\$(FileDate)\$ (FileTi</code>

Current Report Option	Equivalent Expression
Sequence File Name to Report File Name + New UUT Report File for Each UUT + Add Date and Time to File Name	

The following table lists example expressions for common tasks using the Sequential process model.

Task	Equivalent Expression
Save reports in a directory that uses the sequence filename	<code>C:\\\$(ClientFileName)\\Report.\$(FileExtension)</code>  Note TestStand overwrites existing files unless you use the \$(Unique) macro or enable the Append if File Already Exists option on the Contents tab of the Report Options dialog box.
Save reports in different directories based on execution status	<code>C:\\\$(ClientFileName)\\\$(UUTStatus)\\Report.\$(FileExtension)</code>  Note TestStand overwrites existing files unless you use the \$(Unique) macro or enable the Append if File Already Exists option.
Generate report	<code>C:\\\$(ClientFileName)\\Report_\$(UUT).\$(FileExtension)</code>

Task	Equivalent Expression
filenames that include the UUT serial number	
Generate report filenames that include the user name	C:\\\$ (ClientFileName) \\Report_ \$ (UserName) . \$ (FileExtension)
Generate report filenames that include the execution status	C:\\\$ (ClientFileName) \\Report_ \$ (UUTStatus) . \$ (FileExtension)



Note NI provides a list of supported macros you can use with expressions to customize reports. Use these macros only in the Specify Report File Path by Expression control on the Report File Pathname tab of the Report Options dialog box and not in any other TestStand expression controls.

Specify Report File Path by Expression

To specify an expression that generates the report file path during execution, select **Specify Report File Path by Expression** from the **File/Directory Options** ring control on the Report File Pathname tab of the Report Options dialog box. When you select this option, the expression controls become visible depending on the process model you use. If you are using the Sequential or Parallel process model, or if you are using the Batch process model and you select ATML Report Document as the report format, only the Report File Path expression control is visible. If you are using the Batch process model and you select a report format other than ATML Report Document, both

the Report File Path and Batch Report File Path expression controls are visible.

For example, to store an XML report file in the `C:\Temp\Reports` directory with the filename stored in a variable `FileGlobals.ReportFileName` in the current process model, use the expression `"C:\\Temp\\Reports\\" + FileGlobals.ReportFileName + ".xml"`.

You can use predefined macros to customize the UUT or Batch Report File Path expression. During an execution, the process model replaces the predefined macros in the UUT or Batch Report File Path expressions according to the options you select. You can use the following macros in the UUT or Batch Report File Path expressions, with some exceptions.



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

- `$(Batch)` —Replaces the macro with the batch serial number. The `$(Batch)` macro creates new Batch Report files for each serial number in the batch.



Note You can use the `$(Batch)` macro only with the Batch process model.

- `$(ClientFileDir)` —Replaces the macro with the directory that contains the client sequence file. For example, if you execute the client sequence saved as `c:\Tests\TestSequences\TestSequence1.seq` using an Execution entry point, `$(ClientFileDir)` specifies the `c:\Tests\TestSequences` directory.



Note If you use the `$(ClientFileDir)` macro without saving the sequence file to disk, the macro evaluates to the `<TestStand>\Temp` directory.

- `$(ClientFileName)` —Replaces the macro with the name of the client sequence file. For example, if you execute the client sequence file saved as `c:\Tests\TestSequences\TestSequence1.seq` using an Execution entry point, the `$(ClientFileName)` macro evaluates to the string

"TestSequence1".



Note If you use the `$(ClientFileName)` macro without saving the sequence file to disk, the report generator replaces the macro with the string "Unsaved Sequence File". You can modify the string by changing the `ModelStrings.ini` language file.

- `$(Desktop)` —Replaces the macro with the directory path of the desktop.
- `$(FileExtension)` —Replaces the macro with a file format extension that corresponds to the report format you specify on the Contents tab of the Report Options dialog box. For example, if you specify the report file format as HTML, the process model replaces the macro with the string "html".
- `$(FileDate)` —Replaces the macro with a string containing the current date in localized format when the report is first saved.
- `$(FileDay)` —Replaces the macro with a string containing the numeric day of the month when the report is first saved.
- `$(FileMonth)` —Replaces the macro with a string containing the numeric month when the report is first saved.
- `$(FileTime)` —Replaces the macro with a string containing the current time in localized format when the report is first saved.
- `$(FileYear)` —Replaces the macro with a string containing the numeric year when the report is first saved.
- `$(PublicDocumentsDir)` —Replaces the macro with the file system directory that contains documents common to all users. A typical path is `C:\Users\Public\Documents` on Microsoft Windows 10/8.1/7.
- `$(StationID)` —Replaces the macro with the station ID.
- `$(TempDir)` —Replaces the macro with the `<TestStandTemp>` directory.
- `$(TestSocket)` —Replaces the macro with `TestSocketIndex` during run time. This macro creates new report files for each test socket.



Note The `$(TestSocket)` macro is invalid in the Batch Report File Path expression.

- `$(TestStandExecutableDir)` —Replaces the macro with the directory of the TestStand executable. For example, if you are using the TestStand Sequence Editor, the `<TestStand>\Bin` directory path replaces this macro.
- `$(TestStandPublicDir)` —Replaces the macro with the `<TestStand Public>`

directory, in which TestStand stores publicly editable files.

- `$(Unique)` —Replaces the macro with a unique numeric value if the file already exists in the path. If you use more than one `$(Unique)` macro in the Report File Path expression, TestStand replaces only the first instance from left with the unique numerical value. For example, if you specify the Report File Path expression as
`"C:\\Temp$(Unique)\\TSReports$(Unique)\\Report$(Unique).xml"`
 and `C:\\Temp\\TSReports\\Report.xml` already exists, the process model saves the report file path at `C:\\Temp0001\\Reports\\Report.xml`.



Note You cannot use the `$(Unique)` macro as a stand-alone directory or filename in the Report File Path or the Batch Report File Path expression.

- `$(UserDocumentsDir)` —Replaces the macro with the My Documents desktop item.
- `$(UserName)` —Replaces the macro with the user name of the currently logged in user.



Note If no user is currently logged in, an empty string replaces the `$(UserName)` macro.

- `$(UUT)` —Replaces the macro with the UUT serial number. This macro creates new UUT report files for each UUT.



Note

- If you use the `$(UUT)` macro and the UUT serial number is the same for two different UUT loops, the new report file might overwrite the older report file when the Append if File Already Exists option on the Contents tab is disabled. To avoid overwriting the file, use the `$(Unique)`, `$(FileTime)`, or `$(FileDate)` macros, or enable the **Append if File Already Exists** option.
- The `$(UUT)` macro is invalid in the Batch Report File Path expression.

- `$(UUTStatus)` —Replaces the macro with the UUT status during run time. When you use the `$(UUTStatus)` macro, TestStand creates separate files for each UUT

status. If the UUT Report File Path expression contains the `$(UUTStatus)` and `$(UUT)` macros, TestStand creates new UUT report files for each UUT status. If the UUT Report File Path expression contains the `$(UUTStatus)` and `$(TestSocket)` macros when using the Parallel or Batch process model, for each test socket index, TestStand creates separate files for each UUT status. If the UUT Report File Path expression contains the `$(UUTStatus)` and `$(Batch)` macros, for each batch serial number, TestStand creates separate files for each UUT status.



Note

- If you use on-the-fly (OTF) reporting with the `$(UUTStatus)` macro in the Report File Path expression, the process model creates the temporary file used during OTF reporting in the `<TestStand>\Temp` directory. The process model copies the temporary file to the correct location when the execution completes.
- If you use the `$(UUTStatus)` macro in the Report File Path expression and execute the sequence file using the Test UUTs Execution entry point, TestStand displays only the complete report for the last UUT status at the end of the execution.
- The `$(UUTStatus)` macro is invalid in the Batch Report File Path expression.

- `$(UUTPartNum)` —Specifies the UUT part number.



Note

- If you use the `$(UUTPartNum)` macro and the UUT part number is the same for two different UUT loops, the new report file might overwrite the older report file when the Append if File Already Exists option on the Contents tab is disabled. To avoid overwriting the file, use the `$(Unique)`, `$(FileTime)`, or `$(FileDate)` macros, or enable the Append if File Already Exists option.
- The `$(UUTPartNum)` macro is invalid in the Batch Report File Path expression control.

Processing Report and Batch Report File Path Expressions

The process model generates the report and batch report file paths by evaluating the Report and Batch Report File Path expressions and then replacing the macros. Since the expression is evaluated first, you can also define macros in variables. When you use the Sequential model, TestStand evaluates all Report File Path expressions in the context of the entry point sequence at run time. When you use the Batch or Parallel process model, TestStand evaluates all Report File Path expressions in the context of the test socket Execution entry point sequence at run time. For example, if you execute the sequence file in the Batch process model using the Test UUTs Execution entry point, the process model evaluates all Report File Path expressions in the context of the Test UUTs - Test Socket Entry Point sequence.

The Report File Path and Batch Report File Path expression controls use the context of the first Execution entry point in the current process model to perform syntax checking on the expressions. The Report Options dialog box stores the Report File Path expression separately for each process model. Therefore, you might see different Report File Path expressions when you switch to a different process model.



Note

- Because the Report File Path is an expression, a "/" precedes all escape characters.
- The static text box in the Report Options dialog box shows the evaluated report file path as you edit the expression. The static text box does not evaluate or replace the `$(UUTStatus)`, `$(UUT)`, `$(TestSocket)`, and `$(Batch)` macros.
- If the output format of the report is an ATML report, report generation will always create a new file for each UUT regardless of Report File Path expression. The ATML report format does not support multiple UUT reports in one file.

The process model determines the report file path at different points during the execution of the process model depending on the macro you use, according to the following guidelines:

- If the Report File Path expression contains the `$(UUT)` macro, the report generator determines the report file path before calling the `MainSequence` callback sequence for each UUT.
- If the Report File Path expression contains the `$(UUTPartNum)` macro, the report generator determines the report file path before calling the `MainSequence` callback sequence for each UUT.
- If the Report File Path expression contains the `$(TestSocket)` macro, the report generator determines the report file path before calling the `MainSequence` callback for the first UUT in the test socket.
- If the Report File Path expression contains the `$(Batch)` macro, the report generator determines the report file path before calling the `MainSequence` callback for the first UUT in the batch.
- If the Report File Path expression contains the `$(UUTStatus)` macro, the report generator determines the report file path after calling the `MainSequence` callback. The `$(UUTStatus)` macro causes the process model to group UUT reports into different files according to their status. For example, if a Report File Path expression contains the `$(UUTStatus)` and `$(TestSocket)` macros, the process model stores all the `PASSED` UUT results for a given test socket in one file and stores all `FAILED` UUT results for a given test socket in another file. The process model keeps track of the UUT files generated for each UUT status and appends them as necessary. The following example illustrates the recommended usage of the `$(UUTStatus)` macro for process models.
 - **Sequential model**—Use the `$(UUTStatus)` macro in the Report File Path expression to generate report files containing UUT reports for UUTs with the same UUT status in the same file. For example, to obtain all `PASSED` results in the file path `"C:\TestStandReports\Report_Passed.xml"` and `FAILED` results in the file path `"C:\TestStandReports\Report_Failed.xml"`, use the expression `"C:\\TestStandReports\\Report_$(UUTStatus).xml"` as the report file path.
 - **Parallel model**—Use the `$(UUTStatus)` and `$(TestSocket)` macros in the Report File Path expression to generate report files containing UUT reports of UUTs with the same UUT status in the same file for each test socket. For example, to obtain for each test socket all `PASSED` results of test socket index 1 in the file path `C:\TestStandReports\TestSocketIndex_1\Report_Passed.xml`, all `FAILED` results of test socket index 1 in the file path

"C:\TestStandReports\ \TestSocketNo_1\

Report_Failed.xml", and so on, use the expression

"C:\ \TestStandReports\ \TestSocketIndex_\$(TestSocket) \ \Report_ as the report file path.

- **Batch Model**—Use the \$(UUTStatus) and \$(Batch) macros in the Report File Path expression to group all UUT reports within the batch that have similar UUT status into one UUT Report file. For example, to obtain for each batch all PASSED results of a batch with batch number 'abc' in the file path "C:\TestStandReports\Batch_abc\Report_Passed.xml", all FAILED results of a batch with batch number 'abc' in the file path "C:\TestStandReports\Batch_abc\Report_Failed.xml", and so on, use the expression "C:\TestStandReports\Batch_\$(Batch)\Report_\$(UUTStatus).xml" as the report file path.

Related concepts:

- [Search Directories](#)
- [Expressions](#)

Related information:

- [Report Options Dialog Box](#)

Customizing the Graph Control in Reports

You can customize the graph control for your reports by editing the stylesheet for the report.

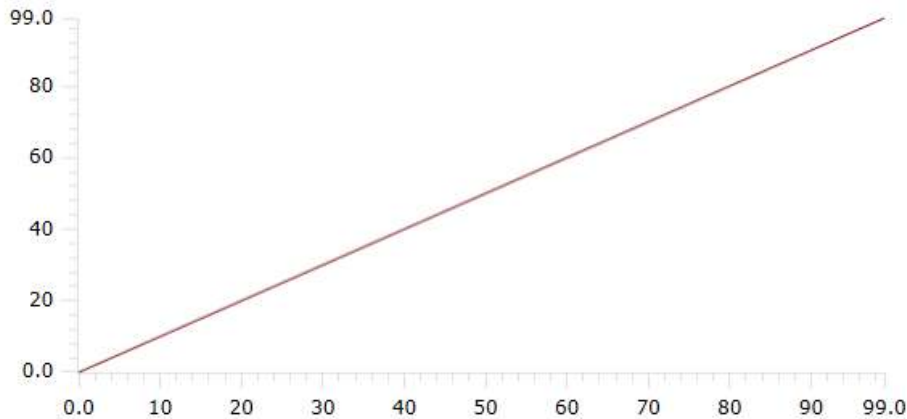
Removing Grid Lines

Complete the following steps to remove grid lines from the graph:

1. Open one of the following files in a text editor:
 - For HTML reports, open <TestStand_Components>\Models\TestStandModels\modelsupport2\c_report.c.
 - For XML and ATML reports, open the stylesheet listed in the Style Sheet field of the Report Options dialog box.
2. Search for the GRID_LINES comment.

3. Set the value of the variable `gridLines` to "grid-lines" to enable grids lines, or to "" to disable grid lines.
4. For HTML reports, open `<TestStand_Components>\Models\TestStandModels\modelsupport2\modelsupport2.prj` in LabWindows/CVI and build `modelsupport2.dll` for the corresponding bitness of TestStand you are using.

The following image shows how the graph looks without grid lines.

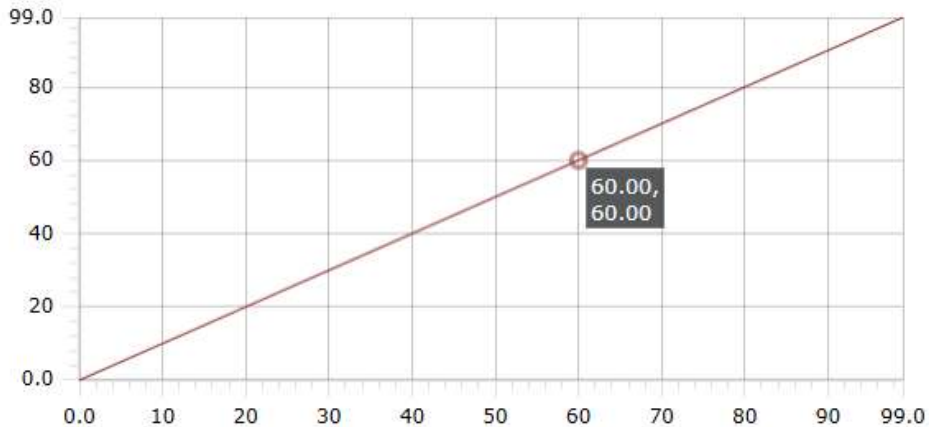


Enabling the Display of Coordinates when Hovering Over a Point

Complete the following steps to enable displaying coordinates when you hover a mouse over a point on the graph:

1. Open one of the following files in a text editor:
 - For HTML reports, open `<TestStand_Components>\Models\TestStandModels\modelsupport2\c_report.c`.
 - For XML and ATML reports, open the stylesheet listed in the Style Sheet field of the Report Options dialog box.
2. Search for the `ENABLE_HOVER` comment.
3. Set the value of the variable `enableHover` to "enable-hover" to enable hovering, or to "" to disable hovering.
4. For HTML reports, open `<TestStand_Components>\Models\TestStandModels\modelsupport2\modelsupport2.prj` in LabWindows/CVI and build `modelsupport2.dll` for the corresponding bitness of TestStand you are using.

The following image shows how the graph looks when hover is enabled.

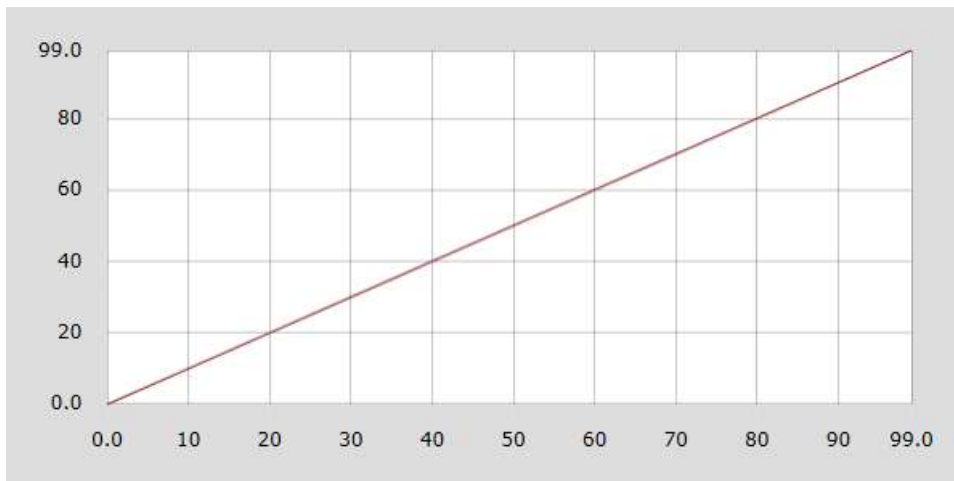


Changing the Background Color of the Graph (Excluding the Plot)

Complete the following steps to change the background color of the area outside the plot of the graph:

1. Open one of the following files in a text editor:
 - For HTML reports, open `<TestStand_Components>\Models\TestStandModels\modelssupport2\c_report.c`.
 - For XML and ATML reports, open the stylesheet listed in the Style Sheet field of the Report Options dialog box.
2. Search for the `CHANGE_BACKGROUND_COLOR` comment.
3. Change the value of `background-color` in the `<ni-cartesian-graph>` tag to the required color name or hex color code. Refer to [HTML Color Codes](#) for more information.
4. For HTML reports, open `<TestStand_Components>\Models\TestStandModels\modelssupport2\modelssupport2.prj` in LabWindows/CVI and build `modelssupport2.dll` for the corresponding bitness of TestStand you are using.

The following image shows how the graph looks when value is set to #dddddd.

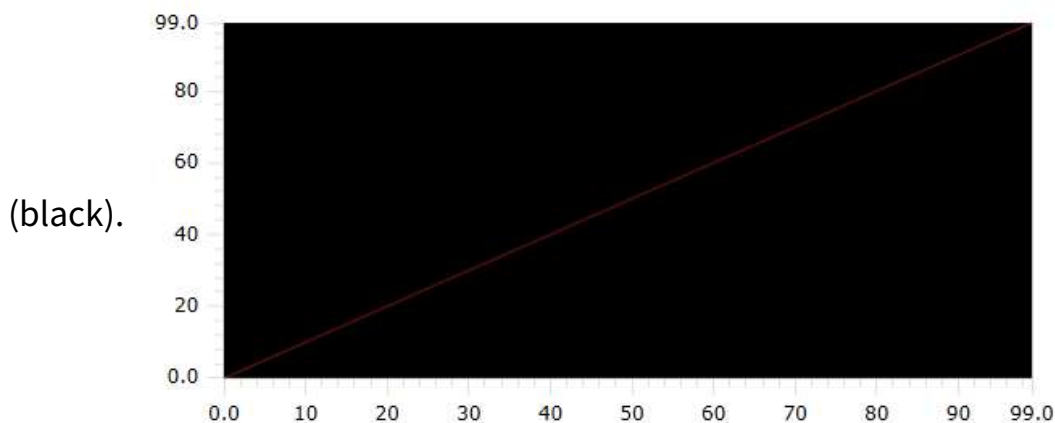


Changing the Background Color of the Plot

Complete the following steps to change the background color of the plot area of the graph:

1. Open `<TestStand_Components>\Models\TestStandModels\GraphControl\TSGraphStyle.css` in a text editor.
2. Search for the `CHANGE_BACKGROUND_COLOR` comment.
3. Change the value of `background-color` to the required color name or hex color code. Refer to [HTML Color Codes](#) for more information.

The following image shows how the graph looks when the value is set to `#000000`



Related information:

- [HTML Color Codes](#)

Failure Chain in Reports

For UUTs that fail, ATML 6.01, ATML 5.0, XML, HTML, and ASCII-text reports include in the report header a failure chain section, which includes the following information:

- The first item in the failure chain table shows the step failure that caused the UUT to fail.
- The remaining items show the Sequence Call steps through which the execution reached the failing step.

In ATML 6.01, ATML 5.0, XML, and HTML reports, each step name in the failure chain links to the section of the report that displays the result for the step.

Batch Reports

When you use the Batch process model, the model generates a batch report in addition to a report for each UUT. The batch report summarizes the results for all the UUTs in the batch. ATML 6.01, ATML 5.0, XML, and HTML reports link to each UUT report.

The ATML 2.02 standard does not support batch reports. The Batch process model generates only UUT reports when you select ATML 2.02 as a report format.

Related concepts:

- [Batch Process Model](#)

Property Flags That Affect Reports

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

The `PropFlags_IncludeInReport` flag specifies to include a property in the report. For properties that hold limit values or output values, use the `PropFlags_IsLimit` and `PropFlags_IsMeasurementValue` flags to selectively exclude limits or output values according to the options you select in the

Report Options dialog box. When you set a reporting flag for an array or container property, TestStand sets the flag for all array elements or subproperties within the container property.

Storing Additional Information for UUTs and Test Stations in Process Models

Where to Access the UUT and StationInfo Variables to Make Changes

You can access the UUT as `Parameters.UUT` and the `StationInfo` as `Parameters.ModelData.StationInfo` in client file callbacks such as `PreUUT` or in process model plug-in entry points such as `Model Plugin - Pre UUT`.



Note You can access the UUT instance for the batch as a whole as `Parameters.ModelData.BatchUUT` in callbacks such as `PreBatch` or in process model plug-in entry points such as `Model Plugins - Pre Batch` plug-in entry point, the default report generators do not use the contents of the `AdditionalData` field of this instance.

UUT Data Type Subproperties

The UUT data type in the default process models includes the subproperties described in the following table for storing UUT information. The `AdditionalData` container subproperty was added in TestStand 2013.

Table 22. UUT Data Type Subproperties

Subproperty	Type	Content	How to Include in Report	How to Include in Database
<code>SerialNumber</code>	String	UUT serial number	The default report generator process model plug-in automatically adds the UUT serial number to a report.	The default schemas for the database automatically add the UUT serial number to the <code>UUT_RESULTS</code> table.
<code>BatchSerialNumber</code>	String	Batch serial number	The default report generator process	The default schemas for the database

Subproperty	Type	Content	How to Include in Report	How to Include in Database
			model plug-in automatically adds the batch serial number to a report.	database l automatically the batch s number to UUT_RES table.
PartNumber	String	UUT part number	The default report generator process model plug-in automatically adds the UUT part number, if specified, to a report.	The default schemas for database l automatically the UUT part number to UUT_RES table.
AdditionalData	Unstructured Container	You can add subproperties of any data type to this container to store additional information associated with a UUT. For example, you can add subproperties to the <code>Locals.UUT.AdditionalData</code> container in the default process model entry points at edit-time, or you can programmatically add subproperties at run-time where the process model passes this variable.	To specify that all subproperties automatically appear in a report, set the IncludeInReport property flag on the <code>AdditionalData</code> property. To specify that only a subset of subproperties automatically appear in a report, set the IncludeInReport property flag on only the subproperties you want to include.	The default schemas for database l do not automatically any additional To log subproper the <code>Additio</code> container, customize schema in Database (dialog box database t database managem system (DE

NI_StationInfo Data Type

The `NI_StationInfo` data type in the default process models includes the subproperties described in the following table for storing station information. The

AdditionalData container subproperty was added in TestStand 2013.

Table 23. NI_StationInfo Data Type Subproperties

Subproperty	Type	Content	How to Include in Report
StationID	String	Station ID	The default generator plugin automatically includes the station ID in the report.
LoginName	String	User login name	The default generator plugin automatically includes the user login name in the report.
AdditionalData	Unstructured Container	You can add subproperties of any data type to this container to store additional information associated with a station. For example, you can add subproperties to the Locals.ModelData.StationInfo.AdditionalData container in the default process model entry points at edit-time, or you can programmatically add subproperties at run-time where the process model passes this variable.	The default generator plugin does not include additional subproperties in the report. To include the subproperties in the report, you must customize the generator plugin.

Related concepts:

- [Model Callbacks in the Parallel Process Model](#)
- [Model Plugin – Pre UUT](#)
- [Model Plugin – Pre Batch](#)
- [Default TestStand Table Schemas](#)
- [UUT_RESULT Table Schema](#)
- [TestStand Database Fundamentals](#)
- [Property Flags That Affect Reports](#)

Escaping Special Characters in Reports

TestStand ATML or XML reports escape certain characters in strings to correctly convert the characters to HTML codes for displaying XML tags in a web browser, as the following table describes. TestStand HTML reports never escape these characters.



Note TestStand ATML or XML reports do not escape these characters when they appear in the `Step.Result.Error.Msg` or `Step.Result.ReportText` properties.

XML Character	Escaped Character
<	<
>	>
&	&
'	'
"	"

Related concepts:

- [ATML](#)
- [XML Reports](#)

On-the-Fly Report Generation

When you enable the **On-The-Fly Reporting** option on the Contents tab of the Report Options dialog box, the process models progressively generate the report concurrently

with the execution instead of waiting until UUT testing completes. The final report TestStand generates is identical to the report the process model generates at the end of execution. You can use on-the-fly reporting for all report formats.

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

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

Configuring On-the-Fly Report Options

When you enable the **Only Display Latest Results** option on the Contents tab of the Report Options dialog box, on-the-fly reporting periodically purges internal data structures while testing the UUT. As a result, the report TestStand displays on the Report pane of the Execution window shows only the results for the steps on-the-fly reporting has not yet purged. The persisted temporary and final report files contain all the step results. TestStand always purges internal data structures when UUT testing completes.



Note The time required to generate reports on-the-fly varies depending on the report format, but you must also consider other benefits of each report format to determine the best format to use for an application. For example, generating an ASCII report on-the-fly might require less time than generating an XML report on-the-fly, but you can more easily parse data from an XML report than you can from an ASCII report. Effectively parsing data from an ASCII report might require that you develop a custom parsing tool.

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



Note If you have configured the process model to use multiple process model plug-ins and you enable the Discard Results or Disable Results When Not Required By Model option in the Model Options dialog box, the process model discards or disables results only when you have configured all process model plug-ins to process results on-the-fly.

On-the-Fly Report Server

In addition, you can use the on-the-fly report server in conjunction with objects of the ReportSection class to create a custom on-the-fly report in any format or layout, including custom report formats. The ProcessOTFStepResults sequence of the reportgen_atml.seq, reportgen_xml.seq, reportgen_html.seq, and reportgen_text.seq report generation sequence files, located in the <TestStand>\Components\Models\TestStandModels directory, calls the on-the-fly report server.

Related concepts:

- [Process Model Architecture](#)
- [Choosing the Appropriate Report Generation Strategy](#)
- [TestStand Directory Structure](#)

Including Binary Strings in Reports

Typically, you cannot store binary data in a string since strings cannot contain zero bytes, and strings must not contain invalid multibyte character set (MBCS) sequences. However, the TestStand LabVIEW Adapter and the TestStand API function SetValBinary support assigning binary values to string variables or properties. TestStand stores binary string values as compressed and encoded binary data using only printable non-MBCS-lead-byte characters.

The default TestStand report generator process model plug-in generates the binary string values in the compressed encoded formats. See the Retrieving Binary String Values from Reports and Databases topic for more information about retrieving binary data from string values.

Related concepts:

- [Retrieving Binary String Values from Reports and Databases](#)

Retrieving Binary String Values from Reports and Databases

The TestStand LabVIEW Adapter and the TestStand API function

`PropertyObject.SetValBinary` support assigning binary values to string variables or properties. TestStand stores binary string values as compressed and encoded binary data using only printable non-MBCS-lead-byte characters. The default TestStand report generator and database logging process model plug-ins store the binary string values in the compressed encoded format.

To retrieve the binary data from a string value in reports or databases, you must first assign the string value to a TestStand string property either by calling the `PropertyObject.SetValString` method or by copying the value into a string property in the TestStand Variables View. Then you must retrieve the binary value using the `PropertyObject.GetValBinary` method on the string property.



Note This method of retrieving binary data requires that the code page of the system that logged the data be the same as the code page of the system retrieving the data.

Displaying Measurement Data as Graphs

TestStand supports displaying one- and two-dimensional measurement array data as a graph in ATML 6.01, ATML 5.0, XML, and HTML reports using the HTML5 graph control, located in the `<TestStand>\Components\Models\TestStandModels\GraphControl` directory. The graph control supports displaying data as a Y plot or as an XY plot. The graph control supports orientation of the array data by rows or columns.

You can add `PropertyObject` attributes to variables you log to specify the layout and orientation of the array data. For example, for parameters, you can add these attributes to local variables you specify in the value expression for each parameter. The graph control uses these `PropertyObject` attributes to interpret the data and display the appropriate plot or plots.

By default, the graph control adds plots using the row-based orientation of the array data. You can add a `PropertyObject` attribute with the `TestStand.DataOrientation` namespace to configure the graph control to interpret the array data by columns.

The type of plot the graph control displays depends on the following factors, as the following table describes:

- The number of dimensions of the array data being displayed.
- The value of the array data `PropertyObject` attribute with the `TestStand.DataLayout` namespace.
- The value of the array data `PropertyObject` attribute with the `TestStand.DataOrientation` namespace.

Dimension of Array Data	DataLayout PropertyObject Attribute Value	DataOrientation PropertyObject Attribute Value	Plot(s) Displayed in the Report
1D	<EmptyString>	<NotApplicable>	Single Y plot. The X coordinates of the plotted points are the array indexes. The Y coordinates of the plotted points are the array data values.
2D	<EmptyString>	<NotUsed>	Multiple Y plots. A plot is created for each row in the array. The X coordinates of the plotted points are the array indexes. The Y coordinates of the plotted points are the array data values.
2D	MultipleY	Row Based	Multiple Y plots. A plot is created for each row in the array. The X coordinates of the plotted points are the array indexes. The Y coordinates of the plotted points are the array data values.
2D	MultipleY	Column Based	Multiple Y plots. A plot is created for each column in the array. The X coordinates of the plotted points are the array indexes. The Y coordinates of the plotted points are the array data values.
2D	SingleX-MultipleY	Row Based	Multiple XY plots. A plot is created for each row in the array. The X coordinates of the

Dimension of Array Data	DataLayout PropertyObject Attribute Value	DataOrientation PropertyObject Attribute Value	Plot(s) Displayed in the Report
			plotted points are the array data values in the first row of the array. The Y coordinates of the plotted points are the array data values of every row in the array except the first row.
2D	SingleX-MultipleY	Column Based	Multiple XY plots. A plot is created for each column in the array. The X coordinates of the plotted points are the array data values in the first column of the array. The Y coordinates of the plotted points are the array data values of every column in the array except the first column.



Note You must use the case-sensitive names for the `TestStand.DataLayout` and `TestStand.DataOrientation` namespaces for ATML 6.01, ATML 5.0, XML, and HTML reports to render graphs correctly.

Related concepts:

- [TestStand Directory Structure](#)
- [PropertyObject Attributes](#)

Displaying Measurement Data in an XML Report

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

- The `TSGraphControl3.dll` ActiveX control, located in the `<National Instruments>\Shared\TestStand` directory, is not installed or registered on the computer.
- The My Computer security zone setting on the Security tab of the Internet Options dialog box on the Microsoft Windows Control Panel is `High`. Microsoft Internet Explorer security settings can prevent web pages from using scripting to create and use an ActiveX control. When you view web pages located on the computer,

Internet Explorer uses the My Computer security zone setting. A security zone setting of `High` prevents scripting from running locally. By default, the settings for the My Computer security zone are not visible in the Internet Options dialog box.

Related concepts:

- [Displaying Measurement Data as Graphs](#)

Including Hyperlinks in Reports

You can include hyperlinks in supported versions of ATML, XML, and HTML reports. TestStand uses the default application you specify for the file format to open the file when you click the link in the report. The web browser returns an error when you click a link for which the file has been moved or no longer exists.

Complete the following steps to create a hyperlink in a report.

1. Create a local variable with a data type of `Path`.
2. Use the absolute path to the file for which you want to include a link in the report as the value of the variable.
3. Create a container attribute named `TestStand`.
4. Create a Boolean attribute named `Hyperlink` in the `TestStand` container attribute.
5. Set the value of the `Hyperlink` attribute to `True` to include a link to the file in the report. If the value of the `Hyperlink` attribute is `False`, the report displays the path as a string, not a link.
6. Log the variable in the report.



Note You must use the case-sensitive names for the `TestStand` and `Hyperlink` attributes for ATML 6.01, ATML 5.0, and XML reports to include links correctly.

When you use the XML Packaging Utility to distribute ATML, XML, and HTML reports that include links, the utility also distributes the target file of the link.

Related concepts:

- [Path](#)
- [Including Hyperlinks in a Report - TDMS File](#)

Comparing Report Format Features

The following table illustrates the supported features among the TestStand report formats.

Table 24. Data Types

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Array	✓	✓	✓	✓	✓
Container*	✓	✓	✓	✓	✓
Number	✓	✓	✓	✓	✓
String	✓	✓	✓	✓	✓
Boolean	✓	✓	✓	✓	✓
Object reference	—	—	✓	—	—



Note * HTML/Text/XML/ATML reports do not log empty containers.

Table 25. Numeric Formats**

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Real	✓	✓	✓	✓	✓
Integer	✓	✓	✓	✓	✓
Unsigned integer	✓	✓	✓	✓	✓
Hexadecimal	✓	✓	✓	✓	✓
Octal	✓	✓	✓	✓	✓
Binary	✓	✓	✓	✓	✓
Custom numeric formats with units (such as %#B123x)	✓	✓	✓	✓	✓



Note ** ATML 6.01, 5.0, and 2.02 reports do not honor some TestStand numeric format options, such as the Use Uppercase Letters option and the Minimum Field Width option. Additionally, the ATML standard defines a custom criteria for formatting numbers.

Table 26. Attributes

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Attributes of attributes	✓	✓	✓	✓	✓
Array of attributes	✓	✓	✓	✓	✓
Container with attributes	✓	✓	✓	✓	✓
Attributes	✓	✓	✓	✓	✓
Attributes in NI_TDMSReference type	✓	✓	✓	✓	✓
Attributes for subproperties of the UUT.AdditionalData container	✓	—	✓	✓	✓

Table 27. Arrays

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Arrays with negative indexes	✓	—	✓	✓	✓
Display empty graph for empty numeric array when you select Insert	—	N/A	—	✓	N/A

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Te
Graph from the Include Arrays ring control					
Empty arrays	✓	✓	✓	✓	✓
Display array index labels	✓	✓ (Only name, not lower or upper bounds)	✓	✓	✓
Display array index labels for empty 2D arrays	Displays [0..Empty]	N/A (Only name, not lower or upper bounds)	Displays [0..Empty]	Displays [0..Empty] [0..Empty]	Di
Arrays of different data types	✓	✓	✓	✓	✓
Display graph for array of numbers with a numeric format other than Real, Integer, or Unsigned	—	N/A	—	✓	N/

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Integer					

Table 28. Containers***

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Displays Hidden property in container	—	—	✓	—	—
Displays property hierarchy when no parent properties have the "Include In Report" flag enabled	✓	—	✓	—	—
Empty structured containers	✓	✓	✓	✓	✓
Empty unstructured containers	✓	✓	✓	✓	✓
Arrays of containers	✓	✓	✓	✓	✓



Note *** Reports do not display empty containers with or without attributes.

Table 29. Comments

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Comments associated with properties	—	—	—	✓	✓

Table 30. Report Features

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Append report to target file if target file already exists	✓	—	✓	✓	✓
Array representation in	✓	—	✓	✓	—

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
graph					
Array representation in table	✓	✓	✓	✓	✓
Result filtering expression	✓	✓	—	✓	✓
On-the-fly	✓	✓	✓	✓	✓
Conserve memory	✓	✓	✓	✓	✓
Test limits	✓	✓	✓	✓	✓
Measurements	✓	✓	✓	✓	✓
Include execution time	✓	—	✓	✓	✓
Include step results	✓	—ATML 2.0 does not display the report when you disable the Include Step Reports option.	✓	✓	✓
Report colors	✓	—	✓	✓	—
Include attributes	✓	✓	✓	✓	✓
Batch reports	✓	—	✓	✓	✓
TestStand-specific properties (such as generating smaller reports)	✓	✓	N/A	N/A	N/A

Table 31. Horizontal Style Sheet

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Expand/collapse Sequence Call step results	✓	✓	—	N/A	N/A
Log parameters	✓	✓	✓	N/A	N/A

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Report color	✓	—	✓	N/A	N/A
Module time	—	—	—	N/A	N/A

Table 32. Report Style Sheet

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Features supported in XML style sheet	✓	N/A	✓	N/A	N/A

Table 33. Expand Style Sheet

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Features supported in XML style sheet	✓	N/A	✓	N/A	N/A

Table 34. Display in Report

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
Parameters and results	✓	✓	✓	✓	✓
Failure chain	✓	—	✓	✓	✓
Background color for status of the measurement elements in Multiple Numeric Test steps	—	N/A	—	✓	N/A
Escaping strings	✓	✓	✓	—	N/A
TD attributes	Step Description inserted using <code><tr:Description></code>	Step Description and Description inserted using	Step Description, Step ID, and	Description appears as attribute in	Description appears as attribute in

Feature	ATML 6.01 or 5.0	ATML 2.02	XML	HTML	Text
	element. Step ID assigned to the ID attribute of the step. Description appears as attribute in generated report. HTML report does not display Step Description or Step ID.	<code><tr:Description></code> element. Step ID assigned to the ID attribute of the step. HTML report does not display Step ID.	Description appear as attributes in generated report. HTML report does not display Step Description or Step ID.	generated report. Report does not display Step Description or Step ID.	generated report. Report does not display Step Description or Step ID.
Variable with white space	Report collapses multiple white spaces to single white space.	Report collapses multiple white spaces to single white space.	Report collapses multiple white spaces to single white space.	Report collapses multiple white spaces to single white space.	Report displays multiple white spaces.
Execution time with a numeric format other than Decimal	Always displays in decimal format	N/A	Displays in specified numeric format	Displays in specified numeric format	Displays in specified numeric format
Empty sequence file	Header and "No Sequence Results Found"	Header and "No Results Found"	Header and "No Sequence Results Found"	Header only	Header only

Related concepts:

- [NI_TDMSReference](#)
- [Choosing the Appropriate Report Generation Strategy](#)

Handling Large Amounts of Data

When a test generates large amounts of data, TestStand might appear to hang while processing the data and generating the report. Use one of the following approaches to handle large amounts of data in your reports.

- Avoid putting large amounts of data directly in the report. Instead, store the data in a large data file, such as TDMS, and provide a link to the file in the report.
- If you want to add a large amount of data to a report without referencing an external file, use a TSR plugin to generate a TSR file when testing. On a different machine or after testing, use the TSR file to generate reports.



Note The TSR format generates a compact raw results file and, if used, requires the TestStand Offline Results Processing Utility to generate the final report.

Related concepts:

- [Including Hyperlinks in Reports](#)

Generating PDF Reports

The report generator can generate a PDF version of all report formats except for ASCII text.

Navigate to the Report Options dialog box for your report, then select **Generate PDF**. TestStand generates PDF reports only at the end of execution even if you enable on-the-fly reporting.



Note

- PDF report generation is limited to reports of less than 80 MB. When the report exceeds 80 MB, PDF report generation is skipped and no error is reported.
- For images to appear correctly in PDF reports, use an absolute path

when you specify an image location in the `Step.Result.ReportText` property.

XML Reports

TestStand generates two types of XML reports:

- XML reports that validate against the TestStand Report schema, `Report.xsd`, which conforms to the XML W3C Schema. `Report.xsd` is located in the `<TestStand>\Components\Models\TestStandModels` directory.
- Automatic Test Markup Language (ATML) reports that validate against the ATML Test Results and Session Information schema, `TestResults.xsd`, which the ATML standard defines. `TestResults.xsd` is located in the `<TestStand>\Components\Models\TestStandModels\ATML\Schemas` directory.

By default, TestStand generates ATML reports. You cannot modify the XML TestStand generates, the TestStand Report schema, or the ATML Test Results and Session Information report schema, but you can customize how TestStand ATML and XML reports appear when displayed in a browser. Additionally, you can write XML report generators to generate custom XML reports.



Note

- All TestStand report generators support generating reports that contain 64-bit integer data types. Signed 64-bit integers use the `%d` format string, and unsigned 64-bit integers use the `%u` format string.
- XML reports might display a table of measurement data instead of a graph under certain conditions.

Related concepts:

- [TestStand Directory Structure](#)
- [ATML](#)

- [ATML Test Results Reports](#)
- [Displaying Measurement Data in an XML Report](#)
- [Comparing Report Format Features](#)

Transforming XML Reports

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

XSLT also usually uses JavaScript, VBScript, XPath, and XQuery technologies along with XSL to transform XML documents. Refer to the W3C website, located at www.w3.org, for more information about these technologies.

Transforming XML Content to HTML Output

The following example XML file associates a style sheet, `samplexStylesheet.xml`:

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version = '1.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
```

```

<xsl:output method="html"/>
<xsl:template match="/">
  <head>
    <title>Generated HTML Output</title>
  </head>
  <body>
    <table border="1">
      <tr>
        <td>Title</td>
        <td>Author</td>
      </tr>
      <tr>
        <td><xsl:value-of select="//title"/></td>
        <td><xsl:value-of select="//author"/></td>
      </tr>
    </table>
  </body>
</xsl:template>
<xsl:stylesheet>

```

The previous example XSLT code generates the following HTML:

```

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

```

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

Title	Author
XSL tutorial	John Smith

Related concepts:

- [XML Report Style Sheets](#)

Distributing XML Reports

Use the XML Packaging Utility, which you can launch in the following ways, to distribute XML reports:

- Select **Tools»Package XML/HTML Files for Distribution** in the TestStand Sequence Editor.
- (Windows 10) Select **Start»National Instruments»TestStand XML Packaging Utility**.

This utility copies the local XML report files you select, including related style sheets, image files, and referenced files, to a destination directory you specify so you can view the XML report on a computer that does not have TestStand installed.

Related concepts:

- [XML Report Style Sheets](#)

XML Report Style Sheets

Apply a style sheet to an XML report to specify how a web browser or external viewer application displays the XML content. Use the **Style Sheet** control on the Contents tab of the Report Options dialog box to select the style sheet to apply to an XML report.

The `<xml-style sheet>` node of an XML report contains the URL path of the style sheet to apply to the XML file, as shown in the following example in which the `<xml-style sheet>` node references the built-in `horizontal.xsl` TestStand style sheet:

```
<?xml-style sheet type="text/xsl" href="C:\Program Files\
National Instruments\TestStand 2014\Components\Models\
TestStandModels\StyleSheets\horizontal.xsl"?>
```

Built-in Style Sheets

TestStand provides the following style sheets, located in the `<TestStand>\Components\Models\TestStandModels\StyleSheets` directory. You can customize the style sheets directly.

- `horizontal.xml`—Displays reports in a horizontal, tabular, indented format with data for each step in one or more rows, as shown in the following figure.

Begin Sequence: MainSequence
C:\Users\Public\Documents\National Instruments\TestStand 2014 (64-bit)\Examples\Demos\Computer Motherboard Test\LabVIEW\Computer Motherboard Test Sequence.seq

Step	Status	Measurement	Units	Limits		Comparison Type
				Low Limit	High Limit	
Simulation Dialog	Done					
Turn Vacuum Table On	Done					
Powerup Test	Passed					
If (True)	Done					

Step	Status	Measurement	Units	Limits		Comparison Type
				Low Limit	High Limit	
CPU Test	Failed					

Begin Sequence: CPU Test
C:\Users\Public\Documents\National Instruments\TestStand 2014 (64-bit)\Examples\Demos\Computer Motherboard Test\LabVIEW\Computer Motherboard Test Sequence.seq

Step	Status	Measurement	Units	Limits		Comparison Type
				Low Limit	High Limit	
Register Test	Passed					
Instruction Set Test	Failed					
Cache Test	Passed					
FPU Test	Passed					

- `report.xml`—Displays reports in a vertical, tabular, indented format with data for each step in multiple rows, as shown in the following figure. Customizable, color-coded rows visually separate relevant information, such as step group,

Begin Sequence: MainSequence
(C:\Users\Public\Documents\National Instruments\TestStand 2014 (64-bit)\Examples\Demos\Computer Motherboard Test\LabVIEW\Computer Motherboard Test Sequence.seq)

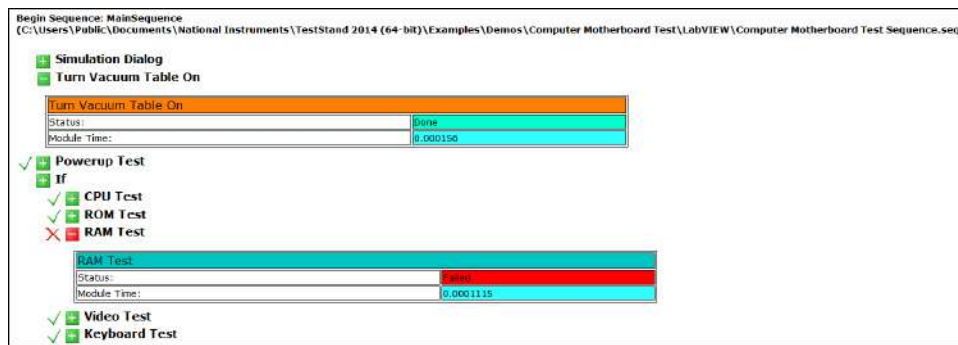
Simulation Dialog	Status:	Done
Module Time:		0.9746602
Turn Vacuum Table On	Status:	Done
Module Time:		0.0001557
Powerup Test	Status:	Passed
Module Time:		0.0001508
If (True)	Status:	Done
Module Time:		
CPU Test	Status:	Failed
Module Time:		0.0130754

Begin Sequence: CPU Test
(C:\Users\Public\Documents\National Instruments\TestStand 2014 (64-bit)\Examples\Demos\Computer Motherboard Test\LabVIEW\Computer Motherboard Test Sequence.seq)

Register Test	Status:	Passed
Module Time:		0.0002056
Instruction Set Test	Status:	Failed
Module Time:		0.0001081
Cache Test	Status:	Passed
Module Time:		0.0001471
FPU Test	Status:	Passed
Module Time:		0.0002233

status, and so on.

- `expand.xml`—Displays reports similar to `report.xml`, with expandable and collapsible sections for each step or for the entire report, as shown in the following figure. The report displays each step result as a separate table.



Using Foreign Characters in XML Report Style Sheets

Using some foreign characters in the XML report style sheets might result in the error "An invalid character was found in text content." To work around this issue, use the HTML code equivalent for the foreign character or add the appropriate encoding attribute to the `<xsl:output>` tag of the style sheet.

Related concepts:

- [TestStand Directory Structure](#)
- [Customizing XML Report Style Sheets](#)
- [Choosing the Appropriate Report Generation Strategy](#)

Sections in XML Report Style Sheets

The TestStand XML report style sheets include a VBScript section, a JavaScript section, and an XSLT section.

VBScript Section

This section contains one function, `GetLocalizedDecimalPoint()`, to return the localized decimal point.

JavaScript Section

This section contains JavaScript functions and variables that the XSLT code in later sections uses to translate the XML document. This section includes the following subsections:

JavaScript Section 1

This section contains the following functions that initialize the global variables and other helper functions:

- `InitStylesheetPath`—Initializes the global path variable, which stores the absolute path of the style sheet.
- `InitFlagGlobalVariables`—Initializes global variables, such as `gIncludeArrayMeasurement`, which acts as a flag to determine how to add array data to the XML output.
- (horizontal.xml) `SetColumnSpanConstant`—Initializes a global variable for the column span of the generated tables.
- (report.xml) and (expand.xml) `InitColorGlobalVariables`—Initializes all color global variables.

JavaScript Section 2

This section contains the following functions to handle localized decimal values:

- `SetLocalizedDecimalPoint`—Sets the localized decimal point to use for all numeric values and stores the localized decimal point in a global variable.
- `ReturnLocalizedDecimalVal_Node`—Replaces the decimal point in the value node with the localized decimal point.

JavaScript Section 3

This section contains the following functions to handle indentation and block levels for the tables in the report:

- `SetResultLevel`—Sets the depth of the results to process.
- `SetBlockLevel`—Sets the block level of the results to process.
- `GetResultLevel`—Returns the depth of the results to process.
- `GetBlockLevel`—Returns the block level of the results to process.
- `InitBlockLevelArray`—Creates the `BlockLevelArray` and initializes the array to 0.
- `ProcessCurrentBlockLevel`—Processes the block level of the step, adds required `<blockquote>` tags for the current block levels, and closes the HTML table if required.
- `GetIndentationString`—Generates an indentation string based on the block level.

- (report.xml) and (horizontal.xml) GetAddTable—Indicates whether you can add a table to the report.
- (report.xml) and (horizontal.xml) SetAddTable—Specifies whether you can add a table to the report.
- (horizontal.xml) GetStdIndentationString—Returns the string that represents the standard indentation.
- (horizontal.xml) GetInitialIndentation—Returns the initial indentation.

JavaScript Section 4

This section contains the following functions to insert looping step results in the report:

- InitLoopArray—Initializes the global array that stores loop index counts.
- BeginLoopIndices—Stores information to process loop index step results.
- (horizontal.xml) and (report.xml) GetLoopIndicesTableEntry—Returns the HTML for the Table Row of the Loop Indices button control.
- TestForStartLoopIndex—Opens the <DIV> and increments the loop stack depth counter for the first loop step result.
- TestForEndLoopIndex—Closes the <DIV> and decreases the loop stack depth counter for the last loop step result.
- InitLoopingInfoArray—Initializes globals information to support looping Sequence Call steps.
- StoreCurrentLoopingLevel—Stores information to support looping Sequence Call steps.
- RestoreLoopingLevel—Restores information to support looping Sequence Call steps.

JavaScript Section 5

This section contains the following functions to insert arrays in the report as graph objects:

- GraphArray—Creates the graph object to display a graph in the report:
- AddElementToGraph—Adds a new numeric value to the graph.
- GetGraphData—Returns the array that contains the data points on the graph.
- GetArrayGraph—Returns the graph object that corresponds to the input array.
- AddArrayToReportAsGraph—Adds an array to the report as a graph.

JavaScript Section 6

This section contains the following functions to insert tables in the report:

- (horizontal.xml) and (report.xml) `BeginTable`—Adds a `<TABLE>` tag to the report.
- (horizontal.xml) and (report.xml) `EndTable`—Adds a `</TABLE>` tag to the report.
- (horizontal.xml) `AddIndentStartTag`—Adds a `<BLOCKQUOTE>` tag to the report.
- (horizontal.xml) `AddIndentEndTag`—Adds a `</BLOCKQUOTE>` tag to the report.
- `IsTableCreationNecessary`—Determines if a step in the current block satisfies the filtering condition and returns `True` to indicate that the block requires a new table. Use this function in filtering steps for Sequence Call steps.

JavaScript Section 7

This section contains the following utility functions:

- (report.xml) and (expand.xml) `GetAbsolutePath`—Returns the absolute path to the directory that contains the style sheet.
- (report.xml) and (expand.xml) `GetModuleTime`—Returns the module time property.
- (report.xml) and (expand.xml) `GetFolderPath`—Converts all `'\'` characters in the path to `'/'`, removes the filename from the input path, and returns the folder path.
- `GetTotalTimeInHHMMSSFormat`—Adds the total execution time in HHMMSS format to the report.
- `AddEndingBlockQuotesForCurrentSequence`—Adds the `</BLOCKQUOTE>` tag to the report after processing a Sequence Call step.
- `GetSerialNumber`—Returns the serial number of the input UUT node or returns `NONE`.
- `GetResultId`—Returns the result ID.
- `GetLoopIndex`—Adds the loop index property of the step to the report.
- `IsNotFlowControlStep`—Returns `True` if the step is not a Flow Control step.
- `GetStepNameAddition`—Returns the report text to add to the step name property for Flow Control steps.
- `RemoveIllegalCharacters`—Adds the `<br>` tag for any newline characters and removes the `\r` character from the text.
- `GetLinkURL`—Returns the complete file URL or only the filename, depending on whether you store absolute or relative paths for the style sheet.
- `AddPropertyToReport`—Determines whether to add a `PropertyObject` to the report based on the `IncludeMeasurement` and `IncludeLimits` report options.

- `IsGraphControlInstalled`—Determines whether the `TSGraphControl2.ocx` ActiveX control is installed.
- (report.xsl) `GetStepBgColor`—Returns the background color for the step.

(expand.xsl) Javascript Section 8

This section contains the following functions to support the expand and collapse functionality:

- `GetIdFromNode`—Creates an ID for the node to use as the ID attribute in the resulting HTML document.
- `GetId`—Calls the `GetIdFromNode` function to create an ID for the input node.
- `GetDIVStyleFromNode`—Returns the style value for a `<DIV>`, depending on the step result status.
- `GetDIVStyle`—Calls the `GetDIVStyleFromNode` function to determine the style.
- `GetStepResultImageFromNode`—Returns the `<IMG>` text for the step result input.
- `GetStepResultImage`—Calls the `GetStepResultImageFromNode` function to retrieve the IMG text.

(expand.xsl) Javascript Section 9

This section contains the following table and filtering condition functions:

- (horizontal.xsl) and (report.xsl) `BeginTableForSequence`—Determines whether to create a table for Sequence Call steps.
- (horizontal.xsl) and (report.xsl) `EndTableForSequence`—Determines whether to add a `</TABLE>` tag for a Sequence Call step.
- `IsTableCreatedForSequence`—Returns True if a new table is required.
- `SetEnableResultFiltering`—Sets or resets the report filtering flag.
- `CheckIfStepSatisfiesFilteringCondition`—Determines whether a step satisfies the filtering condition.
- `CheckIfStepSatisfiesFilteringCondition_node`—Calls the `CheckIfStepSatisfiesFilteringCondition` function to determine whether a step satisfies the filtering condition.

XSLT Section

This section contains XSLT code that transforms XML to HTML. This section includes

the following subsections:

XSLT Section 1

The transformation process to convert XML to HTML starts in this section, which initiates the creation of the HTML page. From this section, the transformation execution flow branches into various templates based on the element or attribute name of the XML tags in the report. This section contains the following subsection:

- (report.xsl) and (expand.xsl)XSLT Section 1.1—JavaScript functions to include with the generated HTML file to support the expand and collapse functionality.

XSLT Section 2

This section contains templates that transform a UUT report. This section contains the following subsections:

- XSLT Section 2.1—Template that processes UUT <Report> elements. This section adds the UUT report header and applies relevant templates that generate the UUT report body.
- XSLT Section 2.2—Templates that include XPath expressions to add data to the UUT report header, such as BatchSerialNumber, TestSocketIndex, SerialNumber and TotalTime.
- XSLT Section 2.3—Template that the TEREResult template calls recursively to process Sequence Call step results. The generation of the UUT report body begins when the transformation process applies this template to the step result that corresponds to the Main sequence. This sequence creates a new table, adds indentation, and applies the TEREResult template to each step result under the <ResultList> element.
- XSLT Section 2.4—Template that adds the sequence file path to the header of the sequence report table. The header consists of the sequence name and the sequence file path where the sequence exists.
- XSLT Section 2.5—Templates that process individual step results, add new rows to the report for each step, and add appropriate step result information. This section uses specific templates that handle steps with PostResults callbacks, looping properties, and Flow Control steps. This section recursively calls the ResultList template in XML Section 2.3 to handle Sequence Call steps.
- XSLT Section 2.6—Templates that add the step name and step execution status to

the report table.

- (horizontal.xml) XSLT Section 2.7—Template that adds the measurement array of Multiple Numeric Limit Test steps to the report.
- XSLT Section 2.8—Template that adds the error message and report text to the report table.
- XSLT Section 2.9—Templates that add the interactive execution number to the report.
- XSLT Section 2.10—Template that adds server information to the report.
- XSLT Section 2.11—Templates that handle summary information for loops of a particular step when looping is enabled for the step or when the user loops or runs only selected steps.
- XSLT Section 2.12—Templates that create the critical failure stack table for failed tests.
- XSLT Section 2.13—Template that adds the module time to the report.

XSLT Section 3

This section contains templates that transform a batch report. From this section, the transformation execution flow branches into various templates based on the element or attribute name of the XML tags in the report. This section contains the following subsections:

- XSLT Section 3.1—Template that processes the batch header table and calls the template that builds the batch report table.
- XSLT Section 3.2—Template that builds the batch report table.
- XSLT Section 3.3—Template that adds data to the batch report table.

XSLT Section 4

This section contains templates that templates in XSLT Section 2 and XSLT Section 3 use. This section contains the following subsections:

- XSLT Section 4.1—Template that processes the StationID and login name.
- XSLT Section 4.2—Template that processes the StartDate and StartTime of the report.

XSLT Section 5

This section contains templates that insert all flagged information and additional results in the report table. This section contains the following subsections:

- XSLT Section 5.1—Templates that process empty cell values, standard indentation, and comparison type.
- XSLT Section 5.2—Template that processes the status color configured in the report options.
- (expand.xml) XSLT Section 5.3—Template that processes the status background color configured in the report options.
- XSLT Section 5.4—Template that inserts flagged values in the report.
- XSLT Section 5.5—Template that adds values in the report if the flag to include it in the report is enabled.
- XSLT Section 5.6—Template that generates a result row to insert in the table.
- XSLT Section 5.7—Template that generates the indentation based on the level.
- XSLT Section 5.8—Template that inserts an array in the report as a table.
- XSLT Section 5.9—Template for the array dimensions.
- XSLT Section 5.10—Template that returns whether the step failure caused the sequence to fail.
- (report.xml) and (expand.xml) XSLT Section 5.11—Template for the step background color configured in the report options.
- (horizontal.xml) XSLT Section 5.12—Template that removes white spaces from a string.
- (horizontal.xml) XSLT Section 5.13—Template that returns whether the step failure caused the sequence to fail from the Step status property.

Related concepts:

- [XML Report Style Sheets](#)

Customizing XML Report Style Sheets

You can use a text editor to customize an XML report style sheet in the following ways:

- Add columns and rows to reports
- Add images and text to headers and footers
- Add Sequence Call step results to reports without indentation

- Change the display format of execution time
- Customize the UUT Report header
- Exclude results from reports based on step properties
- Modify table cell background color based on step status
- Render HTML elements and new line characters in reports



Note Copy the existing style sheets and all support files from the `<TestStand>\Components\Models\TestStandModels\StyleSheets` directory to the `<TestStand Public>\Components\Models\TestStandModels\StyleSheets` directory and make changes to the copies of the files. When you change the name of a style sheet, use the **Style Sheet** control on the Contents tab of the Report Options dialog box to specify the new style sheet when generating XML reports.

Related concepts:

- [XML Report Style Sheets](#)
- [Adding Columns and Rows to an XML Report](#)
- [Adding Images and Text to an XML Report Header or Footer](#)
- [Adding Sequence Call Step Results to an XML Report without Indentation](#)
- [Changing the Display Format of Execution Time in an XML Report](#)
- [Customizing the UUT Report Header in an XML Report](#)
- [Excluding Results from an XML Report Based on Step Properties](#)
- [Modifying Cell Background Color Based on Step Status in an XML Report](#)
- [Rendering HTML Elements and New Line Characters in an XML Report](#)
- [TestStand Directory Structure](#)

Adding Columns and Rows to an XML Report

Before you add extra information to a report, confirm that other XSL templates do not already add to the report the result information you want to include.

horizontal.xsl

To add custom columns to reports using the `horizontal.xsl` file, you must modify the XSL templates and JavaScript code to add a column and then add data to the new

column. Complete the following steps to add an extra StepID column to the right side of the report table using the `horizontal.xsl` file.

1. Open the `horizontal.xsl` file.
2. Search for the `CREATE_EXTRA_COLUMNS` comment in the `BeginTable` function of the JavaScript section.
3. Uncomment the following code to add a new StepID column:


```
<td
  rowspan='2'  valign='bottom'  align='center'
  style='width:30%'><span  style='font-
  size:0.6em'><b>StepID</b></span></td>\n" +
```
4. Search for the `INITIALIZE_COLUMN_SPAN_VARIABLES` comment and increment all the variables by the number of extra columns you are adding to set the column span for result data you add as rows to the report. Using column span information ensures that each row in the report includes the same number of columns and uses the correct formatting. Because you are adding only the StepID column in this example, replace the following code:

```
<xsl:variable name="gSecondColumnSpan5"  select="5"/>
<xsl:variable name="gSecondColumnSpan6"  select="6"/>
<xsl:variable name="gSecondColumnSpan7"  select="7"/>
<xsl:variable name="gSecondColumnSpan8"  select="8"/>
```

with the following code:

```
<xsl:variable name="gSecondColumnSpan5"  select="6"/>
<xsl:variable name="gSecondColumnSpan6"  select="7"/>
<xsl:variable name="gSecondColumnSpan7"  select="8"/>
<xsl:variable name="gSecondColumnSpan8"  select="9"/>
```

5. Search for each of three instances of the `ADD_COLUMN_DATA_1` comment and uncomment the following code for each instance:


```
<td><xsl:value-of
  select="@testReferenceID"/></td>
```

The expression `"@testReferenceID"` specifies the XPath expression to obtain the `testReferenceID` from the XML file. You must customize this XPath expression to match the data you want to add to the additional columns. Refer to the W3Schools website for more information on XPath. Refer to ATML Test Results 6.01 and 5.0 Report Structure for the XML object structure.

6. Repeat steps 2 through 5 to add additional columns.

The following figure shows a modified report with a StepID column for Sequence Call steps using `horizontal.xsl`:

Begin Sequence: MainSequence C:\Users\Public\Documents\National Instruments\TestStand 2014 (64-bit)\Examples\Demos\Computer Motherboard Test\LabVIEW\Computer Motherboard Test Sequence.seq							
Step	Status	Measurement	Units	Limits		Comparison Type	StepID
Simulation Dialog	Done						
Turn Vacuum Table On	Done						
Powerup Test	Passed						
If (True)	Done						

Step	Status	Measurement	Units	Limits		Comparison Type	StepID
CPU Test	Failed						ID#iKb1SeWydeqA9LtZeAS/D

Begin Sequence: CPU Test C:\Users\Public\Documents\National Instruments\TestStand 2014 (64-bit)\Examples\Demos\Computer Motherboard Test\LabVIEW\Computer Motherboard Test Sequence.seq							
Step	Status	Measurement	Units	Limits		Comparison Type	StepID
Register Test	Passed						
Instruction Set Test	Failed						
Cache Test	Passed						
FPU Test	Passed						

End Sequence: CPU Test

report.xsl and expand.xsl

Complete the following steps to add an extra row with StepID information under each step in the report table using the `report.xsl` file and the `expand.xsl` file.

1. Open the `report.xsl` or `expand.xsl` file.
2. Search for the `ADD_EXTRA_ROWS` comment.
3. Uncomment the following code after the `ADD_EXTRA_ROWS` comment:

```
<tr>
  <td>
    <xsl:attribute name="style">border-color:<xsl:value-of
select="$reportOptions/Prop[2Name='Colors']/Prop
[@Name='TableBorder']/Value"/></xsl:attribute>
    <span style='font-size:82%;'>StepID</span>
  </td>
  <td>
    <xsl:attribute name="style">border-color:<xsl:value-of
select="$reportOptions/Prop[2Name='Colors']/Prop
[@Name='TableBorder']/Value"/></xsl:attribute>
    <span style='font-size:82%;'><xsl:value-of
select='./Prop[@Name='TS']/Prop[@Name='StepId']/Value"/></span>
  </td>
</tr>
```

The following figure shows a modified report with a StepID row using `report.xsl`:

Begin Sequence: MainSequence
(C:\Users\Public\Documents\National Instruments\TestStand 2014 (64-bit)\Examples\Demos\Computer Motherboard Test\LabVIEW\Computer Motherboard Test Sequence.seq)

Simulation Dialog	
Status:	Done
Module Time:	17.5329356
StepID	ID#:YzaJ1dw+dEi6/eWhmGMhxD
Turn Vacuum Table On	
Status:	Done
Module Time:	0.0001791
StepID	ID#:vrrgs9A0KUS1GHMKCmg17D
Powerup Test	
Status:	Passed
Module Time:	0.000124
StepID	ID#:Jf9NBpHPeQkg9XEzQ6tA
If {True}	
Status:	Done
StepID	ID#:G3y4++L0XU+RESVbywUNVB

The following figure shows a modified report with a StepID row using `expand.xml`:

Begin Sequence: MainSequence
(C:\Users\Public\Documents\National Instruments\TestStand 2014 (64-bit)\Examples\Demos\Computer Motherboard Test\LabVIEW\Computer Motherboard Test Sequence.seq)

Simulation Dialog	
Status:	Done
Module Time:	1.9852463
StepID	ID#:YzaJ1dw+dEi6/eWhmGMhxD
Turn Vacuum Table On	
Status:	Done
Module Time:	3.62e-005
StepID	ID#:vrrgs9A0KUS1GHMKCmg17D
Powerup Test	
Status:	Passed
Module Time:	4.68e-005
StepID	ID#:Jf9NBpHPeQkg9XEzQ6tA

Related concepts:

- [ATML Test Results 6.01 and 5.0 Report Structure](#)

Related information:

- [W3Schools](#)

Adding Images and Text to an XML Report Header or Footer

You can modify the XSL code that generates report headers and footers. Complete the following steps to add images and text to report headers and footers using the `horizontal.xml` file, `report.xml` file, or `expand.xml` file.

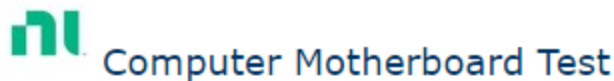
1. Open the `.xml` file you want to change.
2. Remove the comment tags from around the following lines of code after the `ADD_HEADER_INFO` comment to add a company logo to the header and add a Computer Motherboard Test header to the report: `<img src =`

'c:\Images\CompanyLogo.jpg' />Computer Motherboard Test
The HTML tag provides additional functionality, such as aligning the image to the left, right, or center. Refer to the W3Schools website, located at www.w3schools.com, for more information about the tag.

3. Search for the `ADD_FOOTER_INFO` comment and remove the comment tags from around the following line of code before the closing `</body>` tag to add the text **TestStand Generated Report** to the footer of the report:TestStand Generated Report
4. Add `CompanyLogo.jpg` to the `C:\Images` directory.

horizontal.xsl

The following figure shows a modified report header using `horizontal.xsl`:



UUT REPORT

STATION ID	INRD-VS
SERIAL NUMBER	NONE
DATE	Monday, September 23, 2024
TIME	6:10:35 PM
OPERATOR	administrator
EXECUTION TIME	2.3967031 seconds
NUMBER OF RESULTS	17
UUT RESULT	Passed

The following figure shows a modified report footer using `horizontal.xsl`:

END UUT REPORT

TestStand Generated Report

report.xml

The following figure shows a modified report header using `report.xml`:



UUT REPORT

STATION ID	INRD-VS
SERIAL NUMBER	NONE
DATE	Monday, September 23, 2024
TIME	6:11:35 PM
OPERATOR	administrator
EXECUTION TIME	1.2194655 seconds
NUMBER OF RESULTS	17
UUT RESULT	Passed

The following figure shows a modified report footer using `report.xml`:

END UUT REPORT

TestStand Generated Report

expand.xml

The following figure shows a modified report header using `expand.xml`:



Computer Motherboard Test



Expand/Collapse All Step Results

UUT REPORT

STATION ID	INRD-VS
SERIAL NUMBER	NONE
DATE	Monday, September 23, 2024
TIME	6:11:51 PM
OPERATOR	administrator
EXECUTION TIME	1.8322166 seconds
NUMBER OF RESULTS	17
UUT RESULT	Passed

The following figure shows a modified report footer using `expand.xml`:

END UUT REPORT

TestStand Generated Report

Adding Sequence Call Step Results to an XML Report without Indentation

None

horizontal.xml

Change the value of the global JavaScript variable `gIndentTables` from `True` to `False` in the `horizontal.xml` file to disable indentation of the Sequence Call step result tables in the report.



Note Flow Control steps continue to indent even with this customization.

The following figure shows a modified report that includes Sequence Call steps without indentation using `horizontal.xml`:

Begin Sequence: MainSequence (C:\Demo\C\computer.seq)						
Step	Status	Measurement	Units	Limits		
				Low Limit	High Limit	Comparison Type
Simulation Dialog	Done					
Turn Vacuum Table On	Done					
Powerup Test	Passed					
If {True}	Done					

Step	Status	Measurement	Units	Limits		
				Low Limit	High Limit	Comparison Type
CPU Test	Failed					

Begin Sequence: CPU Test (C:\Demo\C\computer.seq)						
Step	Status	Measurement	Units	Limits		
				Low Limit	High Limit	Comparison Type
Register Test	Passed					
PassFail:	True					
Instruction Set Test	Failed					

report.xml

Complete the following steps in the `report.xml` file to disable indentation of the Sequence Call step result tables in the report.

1. Open the `report.xml` file.
2. Search for all instances of the `REMOVE_INDENTATION` comment.
3. Comment out `<xsl:value-of`
`select="user:SetIndentationLevel (user:GetIndentationLevel ()`
`+ 1) "/>` and `<xsl:value-of`
`select="user:SetIndentationLevel (user:GetIndentationLevel ()`
`- 1) "/>` by replacing the code with `<!-- <xsl:value-of`
`select="user:SetIndentationLevel (user:GetIndentationLevel ()`
`+ 1) "/> -->` and `<!-- <xsl:value-of`
`select="user:SetIndentationLevel (user:GetIndentationLevel ()`
`- 1) "/> -->`, respectively.



Note Flow Control steps continue to indent even with this customization.

The following figure shows a modified report that includes Sequence Call steps without indentation using `report.xml`:

If {True}	
Status:	Done
CPU Test	
Status:	Failed
Module Time:	0.0725017
Begin Sequence: CPU Test (C:\Demo\C\computer.seq)	
Register Test	
Status:	Passed
PassFail:	True
Module Time:	0.0002981
Instruction Set Test	
Status:	Failed
PassFail:	False

expand.xsl

Complete the following steps in the `expand.xsl` file to disable indentation of the Sequence Call step result tables in the report.

1. Open the `expand.xsl` file.
2. Search for all instances of the `REMOVE_INDENTATION` comment.
3. Comment out `<xsl:value-of select="user:SetIndentationLevel (user:GetIndentationLevel () + 1) "/>` and `<xsl:value-of select="user:SetIndentationLevel (user:GetIndentationLevel () - 1) "/>` by replacing the code with `<!-- <xsl:value-of select="user:SetIndentationLevel (user:GetIndentationLevel () + 1) "/> -->` and `<!-- <xsl:value-of select="user:SetIndentationLevel (user:GetIndentationLevel () - 1) "/> -->`, respectively.



Note Flow Control steps continue to indent even with this customization.

The following figure shows a modified report that includes Sequence Call steps without indentation using `expand.xsl`:

✖ CPU Test

CPU Test	
Status:	Failed
Module Time:	0.0725017

✖ Begin Sequence: CPU Test
(C:\Demo\C\computer.seq)

✔ ROM Test

ROM Test	
Status:	Passed
PassFail:	True
Module Time:	0.0003327

Changing the Display Format of Execution Time in an XML Report

XSL and JavaScript code generate the HTML code for the execution time information, for which the default display is in seconds. Complete the following steps to display the execution time information in `Hour:Minutes:Seconds` format.

1. Open the `.xsl` file you want to change.
2. Search for the `CHANGE_TOTAL_TIME_FORMAT` comment.
3. Modify the `TotalTime` XSL template to use a different display format for execution time.
 - a. Remove the comment tags from around the following XSL template and add comment tags around the existing `TotalTime` template in the `horizontal.xsl` file and in the `report.xsl` file:


```
<xsl:template match="Prop[@Name='TotalTime']">
  <xsl:param name="reportOptions"/>
  <xsl:value-of disable-output-escaping="yes"
    select="user:GetTotalTimeInHHMMSSFormat (Value)"/>
</xsl:template>
```
 - b. Remove comment tags from around the following XSL template and add comment tags around the existing `TotalTime` template in the `expand.xsl` file:

```
<xsl:template match="Prop[@Name='TotalTime']">
  <xsl:param name="includeStepResults"/>
  <xsl:param name="includeTimes"/>
  <xsl:param name="useLocalizedDecimalPoint"/>
  <xsl:value-of disable-output-escaping="yes"
select="user:GetTotalTimeInHHMMSSFormat (Value)"/>
</xsl:template>
```

horizontal.xsl

The following figure shows the modified execution time in a report using `horizontal.xsl`:

UUT Report	
Station ID	DDT-6KBXDX1
Serial Number	NONE
Date	Tuesday, June 03, 2014
Time	11:24:57 AM
Operator	administrator
Execution Time	00:00:01
Number of Results	26
UUT Result	Failed

report.xsl

The following figure shows the modified execution time in a report using `report.xsl`:

UUT Report

Station ID:	DDT-6KBXDX1
Serial Number:	NONE
Date:	Tuesday, June 03, 2014
Time:	11:16:47 AM
Operator:	administrator
Execution Time:	00:00:01
Number of Results:	26
UUT Result:	Failed
Report Type:	UUT
Failure Chain:	

expand.xml

The following figure shows the modified execution time in a report using `expand.xml`:

 **Expand/Collapse All Step Results**

UUT Report

Station ID:	DDT-6KBXDX1
Serial Number:	NONE
Date:	Tuesday, June 03, 2014
Time:	11:21:10 AM
Operator:	administrator
Execution Time:	00:00:01
Number of Results:	26
UUT Result:	Failed
Failure Chain:	

Customizing the UUT Report Header in an XML Report

You can add columns and corresponding data to the UUT Report header in reports that

use the `horizontal.xml`. You can add rows and corresponding data to the UUT Report header in reports that use the `report.xml` or `expand.xml`.

Complete the following steps to add extra columns or rows to the report header.

1. Open the `.xml` file you want to change.
2. Complete the following steps in the `horizontal.xml`, `report.xml`, and `expand.xml` files to create an extra column and the corresponding data in the UUT header table to show the type of report generated.
 - a. Search for the `CREATE_UUTHEADER_INFO` comment.
 - b. Remove the comment tags from around the following code that appears after the `CREATE_UUTHEADER_INFO` comment:

```
<tr valign="top">
  <td style="white-space:nowrap">
    <span style='font-size:0.6em'>
      <b>Type of Report</b>
    </span>
  </td>
  <td style='white-space:nowrap'>
    <span>
      <xsl:attribute name="style">font-size:0.6em;</xsl:attribute>
      <xsl:value-of select="@Type"/>
    </span>
  </td>
</tr>
```

horizontal.xml

The following figure shows a modified report header showing the report type using `horizontal.xml`:

UUT Report	
Station ID	DDT-6KBXDX1
Serial Number	NONE
Date	Tuesday, June 03, 2014
Time	11:27:32 AM
Operator	administrator
Execution Time	00:00:02
Number of Results	26
UUT Result	Failed
Type of Report	UUT

report.xsl

The following figure shows a modified report header showing the report type using `report.xsl`:

UUT Report	
Station ID:	DDT-6KBXDX1
Serial Number:	NONE
Date:	Tuesday, June 03, 2014
Time:	11:32:02 AM
Operator:	administrator
Execution Time:	00:00:01
Number of Results:	26
UUT Result:	Failed
Report Type:	UUT
Failure Chain:	

expand.xsl

The following figure shows a modified report header showing the report type using `expand.xsl`:

+
Expand/Collapse All Step Results

UUT Report

Station ID:	DDT-6KBXDX1
Serial Number:	NONE
Date:	Tuesday, June 03, 2014
Time:	11:36:35 AM
Operator:	administrator
Execution Time:	00:00:02
Number of Results:	26
UUT Result:	Failed
Type of Report:	UUT
Failure Chain:	

Excluding Results from an XML Report Based on Step Properties

Complete the following steps to exclude results of specific steps based on step properties.

1. Open the .xsl file you want to change.
2. Search for the XSLT variable `gEnableResultFiltering`.
3. Set the variable to 1 to enable result filtering. You can set the variable to 0 to disable result filtering.
4. Search for the string `ADD_STEP_FILTERING_CONDITION`.

The filtering condition in the style sheets includes two expressions. The first expression defines an XPath expression for the filter condition. The second expression performs the filtering based on the filter condition. The style sheets include the following expressions to include only `Passed` steps in the report:

```
filteringCondition = node.selectSingleNode("Prop[@Name='Status']/Value");
if (filteringCondition.text == 'Passed')
```

Refer to the World Wide Web Consortium (W3C) website, located at www.w3.org, for more information about XPath.

The node you pass in as a parameter to the `CheckIfStepSatisfiesFilteringCondition_node` function corresponds to the `<TResult>` node of the step. The report does not include steps under a Sequence Call step when the Sequence Call step itself does not meet the filtering conditions. The report always includes the result of the MainSequence callback step, even if the step does not meet the filtering conditions you set. To include all the steps in a sequence file that satisfy the filtering condition, use a condition to check Sequence Call step types regardless of step status.

Use the following code to exclude Skipped steps:

```
var filteringCondition = node.selectSingleNode("Prop[@Name='Status']/Value");
if (filteringCondition.text != 'Skipped')
```

Use the following code to include only Sequence Call steps or Passed or Done steps:

```
var filteringCondition = node.selectSingleNode("Prop[@Name='Status']/Value");
var stepTypeCondition = node.selectSingleNode("Prop[@Name = 'TS']/Prop[@Name = 'StepType']/Value");
var stepTypeConditionString;
if (stepTypeCondition)
    stepTypeConditionString = stepTypeCondition.text;
else stepTypeConditionString = 'SequenceCall'; //To handle psuedosequencecall steps
if ((filteringCondition.text == 'Passed' || filteringCondition.text == 'Done') ||
    stepTypeConditionString == 'SequenceCall')
```

horizontal.xsl

The following figure shows a filtered report using `horizontal.xsl`:

Begin Sequence: MainSequence (C:\Demo\C\computer.seq)						
Step	Status	Measurement	Units	Limits		
				Low Limit	High Limit	Comparison Type
Powerup Test	Passed					

Step	Status	Measurement	Units	Limits		
				Low Limit	High Limit	Comparison Type
ROM Test	Passed					
PassFail:	True					
Video Test	Passed	5		0	10	GTLT(> <)
Keyboard Test	Passed	6		5		GT(>)

report.xsl

The following figure shows a filtered report using `report.xsl`:

Begin Sequence: MainSequence (C:\Demo\C\computer.seq)	
Powerup Test	
Status:	Passed
Module Time:	0.0003394
ROM Test	
Status:	Passed
PassFail:	True
Module Time:	0.0003327
Video Test	
Status:	Passed
Numeric:	5
Limits:	
Low:	0
High:	10
Comp:	GTLT
Module Time:	0.0003475
Keyboard Test	
Status:	Passed
Numeric:	6
Limits:	
Low:	5

expand.xsl

The following figure shows a filtered report using `expand.xsl`:

Begin Sequence: MainSequence
(C:\Demo\C\computer.seq)

✓ ☐ Powerup Test

Powerup Test	
Status:	Passed
Module Time:	0.0003394

✓ ☐ ROM Test

ROM Test	
Status:	Passed
PassFail:	True
Module Time:	0.0003327

✓ ☐ Video Test

Video Test	
Status:	Passed
Numeric:	5
Limits:	
Low:	0
High:	10

Modifying Cell Background Color Based on Step Status in an XML Report

None

horizontal.xsl

Complete the following steps to modify the background color of the step status cell using the `horizontal.xsl` file.

1. Open the `horizontal.xsl` file.
2. Search for the `ADD_IMG_STATUS` comment.
3. Replace the `<td valign="middle" align="center">` line of code with the following code to set the background color of step result cells to yellow for Passed steps:

```
<xsl:variable name="backgroundColor">
  <xsl:choose>
    <xsl:when test="Value = 'Passed'">#FFFF00</xsl:when>
```

```

        <xsl:otherwise>#FFFFFF</xsl:otherwise>
    </xsl:choose>
</xsl:variable>
<td valign="middle" align="center" bgcolor="{ $backgroundColor }">

```

Optionally, you can add `Failed.jpg` to the `C:\Images` directory and add the following lines of code after the `<td valign="middle" align="center">` line of code to add an image next to the UUT status text for Failed steps:

```

<xsl:if test="Value = 'Failed'">
    <img src = "C:\Images\Failed.jpg"/>
</xsl:if>

```

4. Search for each of three instances of the `ADD_COLUMN_DATA_1` comment and uncomment the following code for each instance:

```
<td><xsl:value-of select="@testReferenceID"/></td>
```

The expression `"@testReferenceID"` specifies the XPath expression to obtain the `testReferenceID` from the XML file. You must customize this XPath expression to match the data you want to add to the additional columns. Refer to the W3Schools website for more information on XPath. Refer to ATML Test Results 6.01 and 5.0 Report Structure for the XML object structure.

The following figure shows a modified report with yellow cells for steps with a `Passed` status using `horizontal.xsl`:

Begin Sequence: MainSequence (C:\Demo\C\computer.seq)						
Step	Status	Measurement	Units	Limits		
				Low Limit	High Limit	Comparison Type
Simulation Dialog	Done					
Turn Vacuum Table On	Done					
Powerup Test	Passed					
If {True}	Done					

Step	Status	Measurement	Units	Limits		
				Low Limit	High Limit	Comparison Type
CPU Test	Failed					

Begin Sequence: CPU Test (C:\Demo\C\computer.seq)						
Step	Status	Measurement	Units	Limits		
				Low Limit	High Limit	Comparison Type
Register Test	Passed					
PassFail:	True					
Instruction Set Test	Failed					
PassFail:	False					
Cache Test	Passed					
PassFail:	True					

report.xsl and expand.xsl

Complete the following steps to modify the background color of the step status cell using the `report.xsl` file and the `expand.xsl` file.

1. Open the `report.xsl` or `expand.xsl` file.
2. Search for the `ADD_IMG_STATUS` comment.
3. Replace the following code in the `BG_COLOR` attribute:

```
<xsl:call-template name="GetStatusBgColor">
  <xsl:with-param name="colors" select="$colors"/>
  <xsl:with-param name="status" select="Value"/>
</xsl:call-template>
```

with the following code to set the background color of step result cells to yellow for Passed steps. The `<xsl:when test="Value = 'Passed' ">#FFFF00</xsl:when>` expression changes the background color variable to yellow. You can use a custom XPath expression instead.

```
<xsl:choose>
  <xsl:when test="Value = 'Passed' ">#FFFF00</xsl:when>
  <xsl:otherwise>
    <xsl:call-template name="GetStatusBgColor">
      <xsl:with-param name="colors" select="$colors"/>
      <xsl:with-param name="status" select="Value"/>
    </xsl:call-template>
  </xsl:otherwise>
</xsl:choose>
```

```

    </xsl:call-template>
  </xsl:otherwise>
</xsl:choose>

```

Optionally, you can add `Failed.jpg` to the `C:\Images` directory and add the following lines of code after the `WIDTH` attribute and before the `<font>` tag to add an image next to the UUT status text for `Failed` steps:

```

<xsl:if test="Value = 'Failed'">
  <img src = "C:\Images\Failed.jpg"/>
</xsl:if>

```

The following figure shows a modified report with yellow cells for steps with a `Passed` status using `report.xsl`:

Powerup Test	
Status:	Passed
Module Time:	0.0003394
If {True}	
Status:	Done
CPU Test	
Status:	Failed
Module Time:	0.0725017
Begin Sequence: CPU Test (C:\Demo\C\computer.seq)	
Register Test	
Status:	Passed
PassFail:	True
Module Time:	0.0002981

The following figure shows a modified report with yellow cells for steps with a `Passed` status using `expand.xsl`:

✓ 	Powerup Test						
<table border="1"> <tr> <td colspan="2">Powerup Test</td></tr> <tr> <td>Status:</td><td>Passed</td></tr> <tr> <td>Module Time:</td><td>0.0003394</td></tr> </table>		Powerup Test		Status:	Passed	Module Time:	0.0003394
Powerup Test							
Status:	Passed						
Module Time:	0.0003394						
	If						
<table border="1"> <tr> <td colspan="2">If {True}</td></tr> <tr> <td>Status:</td><td>Done</td></tr> </table>		If {True}		Status:	Done		
If {True}							
Status:	Done						
✗ 	CPU Test						
<table border="1"> <tr> <td colspan="2">CPU Test</td></tr> <tr> <td>Status:</td><td>Failed</td></tr> <tr> <td>Module Time:</td><td>0.0725017</td></tr> </table>		CPU Test		Status:	Failed	Module Time:	0.0725017
CPU Test							
Status:	Failed						
Module Time:	0.0725017						

Related concepts:

- [ATML Test Results 6.01 and 5.0 Report Structure](#)

Related information:

- [W3Schools](#)

Rendering HTML Elements and New Line Characters in an XML Report

Complete the following steps to ensure a report renders HTML elements and new line characters using the `horizontal.xsl`, `report.xsl`, or `expand.xsl` files.

1. Open the `.xsl` file you want to change.
2. Search for the `Customization variables comment`.
3. Search for the `gDisplayNewLineAndHTMLElementInStringProperty` variable and change the value of the `select` attribute from `"false"` to `"true"`, which results in the following line of code:

```
<xsl:variable
name="gDisplayNewLineAndHTMLElementInStringProperty"select="true()" />
```


For example, if you create a local string variable named `StringContainingHTMLElementAndNewLine` and specify `Locals.StringContainingHTMLElementAndNewLine="<b>Hello\nWorld</b>"` in the `TestStand` expression, executing the sequence file generates a modified report.

horizontal.xsl

The following figure shows the modified report using `horizontal.xsl`:

Step	Status	Measurement	Units	Limits		
				Low Limit	High Limit	Comparison Type
Additional Results	Done					
StringContainingHTMLElementAndNewLine:	Hello World					

report.xsl

The following figure shows the modified report using `report.xsl`:

Additional Results	
Status:	Done
StringContainingHTMLElementAndNewLine:	Hello World

expand.xsl

The following figure shows the modified report using `expand.xsl`:

Additional Results	
Status:	Done
StringContainingHTMLElementAndNewLine:	Hello World

ATML Test Results Reports

The ATML Test Results and Session Information schema provides the definition for the data collected when you execute a test or tests of a UUT using test procedures in an automated test environment, including the measured values, pass/fail results, and accompanying data, such as test operator, station information, environmental conditions, and so on. ATML Test Results is a component standard of IEEE 1636

Software Interface for Maintenance Information Collection Analysis (SIMICA).

TestStand generates XML reports that validate against the following versions of the Test Results and Session Information schema the ATML standard defines:

- Approved version 6.01
- Unapproved version 5.0
- Approved version 2.02

Use the **Report Format** ring control on the Contents tab of the Report Options dialog box to select the version you want to use.

The `<TestStand>\Components\Models\TestStandModels\ATML\Schemas` directory contains copies of versions 6.01, 5.0, and 2.02 of the schema.



Note No plans exist to approve version 5.0 of the ATML Test Results and Session Information schema.

You can customize an ATML report TestStand creates by implementing a DLL that calls one or more exported extension functions.

Known Issues with ATML Report Generators

- When a step containing limits has a run mode set to force failure, TestStand does not evaluate the comparator for the limits. This results in an empty comparator attribute value in the generated report XML, which may cause schema validation errors.
- ATML 2 does not support using an EqualityComparisonOperator attribute that is case sensitive. Therefore, ATML 2 reports always display IgnoreCase regardless of which comparison type you select in a String Value Test step.

Related concepts:

- [ATML](#)
- [Comparing Report Format Features](#)
- [TestStand Directory Structure](#)
- [Customizing the ATML Test Results Report Generator](#)

- [Choosing the Appropriate Report Generation Strategy](#)

ATML

Automatic Test Markup Language (ATML) is an emerging standard in military and aerospace industries for sharing data among different components of a test system. ATML supports test program, test asset, and unit under test (UUT) interoperability within an automatic test environment. ATML accomplishes this through a standard XML schema for exchanging UUT test and diagnostic information among components of the test system.

ATML specifies standards for test environments that encompass the total product life cycle and defines an integrated set of test-related information that supports the information needs of test environments for testing applications.

ATML standards focus on the following areas:

- Diagnostics
- Instrument Description
- Test Adapter
- Test Configuration
- Test Description
- Test Results and Session Information
- Test Station
- UUT Description

Purpose

ATML is intended to accomplish the following objectives:

- Facilitate the communication, sharing, and reuse of product design and test information for the purpose of testing the product.
- Facilitate test program set (TPS) portability and interoperability.
- Facilitate instrument interchangeability.
- Facilitate the development, integration, and use of test software and test software development tools.
- Support the application of integrated diagnostics.
- Support modular software architectures based on a framework that supports

reusable software products.

Refer to the ATML website for more information about ATML.

Related concepts:

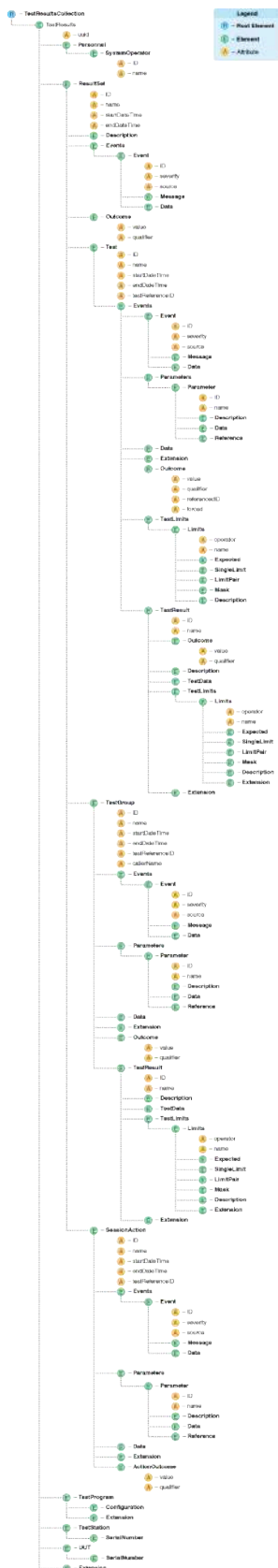
- [ATML Test Results Reports](#)

ATML Test Results 6.01 and 5.0 Report Structure

The TestStand process models use the ATML Test Results (TR) report generator, `ATMLSupport.dll`, located in the `<TestStand>\Components\Models\TestStandModels` directory, to generate a valid ATML TR report format from TestStand variables. The process models call the following functions the report generator implements and exports:

- `Get_Atml_Report`—Generates a valid XML report in TR format.
- `GenerateUUID`—Generates a universally unique identifier (UUID) for the TR report document, which the `uuid` attribute of the `<TestResults>` TR element references.

The TR report generator generates the following XML structure for an ATML TR 6.01 and 5.0 report:





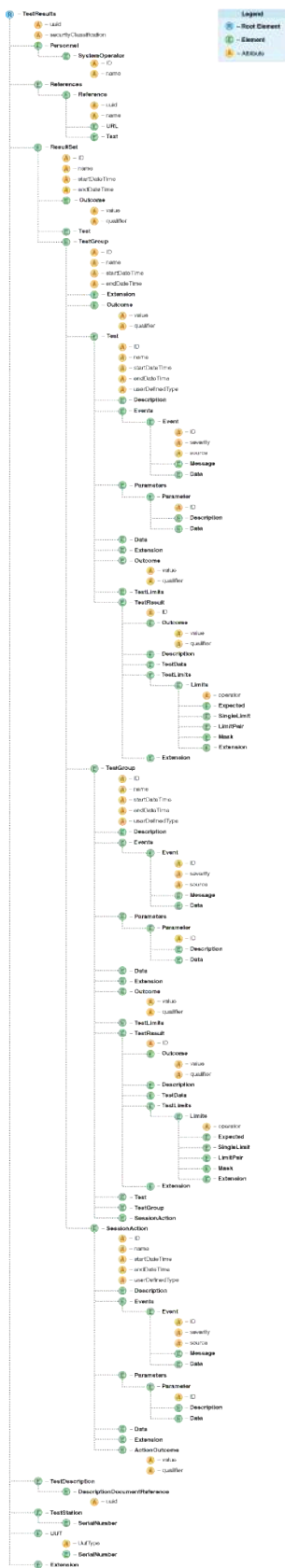
Note The previous figure represents only the elements and attributes TestStand supports.

ATML Test Results 2.02 Report Structure

The TestStand process models use the ATML Test Results (TR) report generator, `ATMLSupport.dll`, located in the `<TestStand>\Components\Models\TestStandModels` directory, to generate a valid ATML TR report format from TestStand variables. The process models call the following functions the report generator implements and exports:

- `Get_Atml_Report`—Generates a valid XML report in TR format.
- `GenerateUUID`—Generates a universally unique identifier (UUID) for the TR report document, which the `uuid` attribute of the `<TestResults>` TR element references.

The TR report generator generates the following XML structure for an ATML TR 2.02 report:





Note The previous figure represents only the elements and attributes TestStand supports.

ATML Report Style Sheets

Apply a style sheet to an ATML report to specify how a web browser or external viewer application displays the ATML content. Use the **Style Sheet** control on the Contents tab of the Report Options dialog box to select the style sheet to apply to an ATML report. You can select a different style sheet depending on the version of the ATML Test Results and Session Information schema against which the report validates.

ATML Test Results 6.01 Reports Style Sheets

TestStand provides the following style sheets, located in the

`<TestStand>\Components\Models\TestStandModels\ATML\StyleSheets` directory, for generating an ATML Test Results 6.01 report. You can customize the style sheets directly.

- **TR6_Expand.xsl**—Displays reports similar to `TR6_report.xsl`, with expandable and collapsible sections for each step or for the entire report.
- **TR6_Horizontal.xsl**—Displays reports in a horizontal, tabular, indented format with data for each step in one or more rows.
- **TR6_report.xsl**—Displays reports in a vertical, tabular, indented format with data for each step in multiple rows. Customizable, color-coded rows visually separate relevant information, such as step group, status, and so on.

ATML Test Results 5.0 Reports Style Sheets

TestStand provides the following style sheets, located in the

`<TestStand>\Components\Models\TestStandModels\ATML\StyleSheets` directory, for generating an ATML Test Results 5.0 report. You can customize the style sheets directly.

- **TR5_Expand.xsl**—Displays reports similar to `TR5_report.xsl`, with expandable and collapsible sections for each step or for the entire report.
- **TR5_Horizontal.xsl**—Displays reports in a horizontal, tabular, indented format

with data for each step in one or more rows.

- **TR5_report.xsl**—Displays reports in a vertical, tabular, indented format with data for each step in multiple rows. Customizable, color-coded rows visually separate relevant information, such as step group, status, and so on.

ATML Test Results 2.02 Reports Style Sheets

TestStand provides the following style sheet, located in the

`<TestStand>\Components\Models\TestStandModels\ATML\StyleSheets` directory, for generating an ATML Test Results 2.02 report. You can customize the style sheet directly.

- **TR_horizontal.xsl**—Displays reports in a concise tabular format with expand and collapse sections. The `TR_horizontal.xsl` style sheet merges the functionality of the `horizontal.xsl` and `expand.xsl` XML report style sheets and displays TestStand PropertyObject attributes for parameters and additional results in the report.

Related concepts:

- [ATML Test Results Reports](#)
- [TestStand Directory Structure](#)
- [Customizing ATML Test Results 6.01 or 5.0 Report Style Sheets](#)
- [Customizing the ATML Test Results 2.02 Report Style Sheet](#)
- [XML Report Style Sheets](#)
- [PropertyObject Attributes](#)

Customizing ATML Test Results 6.01 or 5.0 Report Style Sheets

You can use a text editor to customize a style sheet you can use for a report that validates against version 6.01 or 5.0 of the ATML Test Results and Session Information schema in the following ways:

- Add columns and rows to reports
- Add images and text to headers and footers
- Add step results to reports without indentation
- Add Sequence Call step results to reports without indentation
- Change the display format of execution time

- Customize the UUT Report header
- Modify table cell background color based on step status
- Render HTML elements and new line characters in reports



Note Copy the existing style sheets and all support files from the `<TestStand>\Components\Models\TestStandModels\ATML\StyleSheets` directory to the `<TestStand Public>\Components\Models\TestStandModels\ATML\StyleSheets` directory and make changes to the copies of the files. When you change the name of a style sheet, use the Contents tab of the Report Options dialog box to use the new style sheet when generating ATML reports.

Related concepts:

- [ATML Report Style Sheets](#)
- [ATML Test Results Reports](#)
- [Adding Columns and Rows to an ATML Test Results 6.01 or 5.0 Report](#)
- [Adding Images and Text to an ATML Test Results 6.01 or 5.0 Report Header or Footer](#)
- [Adding Step Results to an ATML Test Results 6.01 or 5.0 Report without Indentation](#)
- [Adding Sequence Call Step Results to an ATML Test Results 6.01 or 5.0 Report without Indentation](#)
- [Changing the Display Format of Execution Time in an ATML Test Results 6.01 or 5.0 Report](#)
- [Customizing the UUT Report Header for an ATML Test Results 6.01 or 5.0 Report](#)
- [Modifying Cell Background Color Based on Step Status in an ATML Test Results 6.01 or 5.0 Report](#)
- [Rendering HTML Elements and New Line Characters in an ATML Test Results 6.01 or 5.0 Report](#)
- [TestStand Directory Structure](#)

Adding Columns and Rows to an ATML Test Results 6.01 or 5.0 Report

Before you add extra information to a report, confirm that other XSL templates do not

already add to the report the result information you want to include.

TR6_Horizontal.xsl or TR5_Horizontal.xsl

To add custom columns to reports using the `TR6_Horizontal.xsl` or `TR5_Horizontal.xsl` file, you must modify the XSL templates to add a column and then add data to the new column. Complete the following steps to add an extra StepID column to the right side of the report table using the `TR6_Horizontal.xsl` or `TR5_Horizontal.xsl` file. You can also copy and add the following code for each new column you want to add.

1. Open the `.xsl` file you want to change.
2. Search for the `<!-- Customization variables-->` comment.
3. Search for the `gNumExtraColumnsToAdd` variable and increment the value of the `gNumExtraColumnsToAdd` variable by the number of columns you want to add. The initial value is 0. For example, to add one extra column, replace the 0 value with 1, which results in the following line of code: `<xsl:variable name="gNumExtraColumnsToAdd" select="1"/>`
4. Search for the `CREATE_EXTRA_COLUMNS` comment and uncomment the following code:

```
<td rowspan="2" valign="bottom" style="text-align:center;" width="10%">
  <b>StepID</b>
</td>
```

5. Search for each of three instances of the `ADD_COLUMN_DATA_1` comment and uncomment the following code for each instance:

```
<td><xsl:value-of select="@testReferenceID"/></td>
```

The expression `"@testReferenceID"` specifies the XPath expression to obtain the `testReferenceID` from the XML file. You must customize this XPath expression to match the data you want to add to the additional columns. Refer to the W3Schools website for more information on XPath. Refer to ATML Test Results 6.01 and 5.0 Report Structure for the XML object structure.

The following figure shows a modified report with a StepID column added for Sequence Call steps using the `TR6_Horizontal.xsl` or `TR5_Horizontal.xsl` file.

Begin Sequence: MainSequence
(C:\Customization\Computer Motherboard Test Sequence.seq)

Step	Status	Measurement	Units	Limits			StepID
				Low Limit	High Limit	Comparison Type	
Simulation Dialog	Done						ID#:EsMLWcS4R0S9I0u0XWrPUC
Turn Vacuum Table On	Done						ID#:JS/FQAK3vEq0MrwZ30/DWD
Powerup Test	Passed						ID#:SCUnEgAAW0yj3vSIBLX6/D
If{True}	Done						ID#:4eEsqOKbqUuns4XRqA2FxC

TR6_report.xml, TR5_report.xml, TR6_Expand.xml, or TR5_Expand.xml

Complete the following steps to add an extra row with StepID information under each step in the report table using the TR6_report.xml, TR5_report.xml, TR6_Expand.xml, or TR5_Expand.xml file.

1. Open the .xml file you want to change.
2. Search for each of three instances of the ADD_EXTRA_ROWS comment and uncomment the following code for each instance:

```
<tr>
  <td>StepID</td>
  <td><xsl:value-of select="@testReferenceID"/></td>
</tr>
```

The following figure shows a modified report with a StepID row added for Sequence Call steps using the TR6_report.xml or TR5_report.xml file.

Simulation Dialog	
Status:	Done
Module Time:	2.172728
StepID	ID#:EsMLWcS4R0S9I0u0XWrPUC
Turn Vacuum Table On	
Status:	Done
Module Time:	0.001665
StepID	ID#:JS/FQAK3vEq0MrwZ30/DWD
Powerup Test	
Status:	Passed
Module Time:	0.002035
StepID	ID#:SCUnEgAAW0yj3vSIBLX6/D
If{True}	
Status:	Done
StepID	ID#:4eEsqOKbqUuns4XRqA2FxC

The following figure shows a modified report with a StepID row added for Sequence

Call steps using the `TR6_Expand.xsl` or `TR5_Expand.xsl` file.

Begin Sequence: MainSequence (C:\Customization\Computer Motherboard Test Sequence.seq)	
Simulation Dialog	
Simulation Dialog	
Status:	Done
Module Time:	2.172728
StepID	ID#:EsMLWcS4R0S9I0u0XWrPUC
Turn Vacuum Table On	
Turn Vacuum Table On	
Status:	Done
Module Time:	0.001665
StepID	ID#:JS/FQAK3vEq0MrwZ30/DWD
Powerup Test	
Powerup Test	
Status:	Passed
Module Time:	0.002035
StepID	ID#:5CUnEgAAW0yj3vSIBLX6/D

Related concepts:

- [ATML Test Results 6.01 and 5.0 Report Structure](#)

Related information:

- [W3Schools](#)

Adding Images and Text to an ATML Test Results 6.01 or 5.0 Report Header or Footer

You can modify the XSL code that generates report headers and footers. Complete the following steps to add images and text to report headers and footers.

1. Open the `.xsl` file you want to change.
2. Search for the `ADD_HEADER_INFO` comment and uncomment the following lines of code to add a company logo and Computer Motherboard Test text to the header of the report:

```
<img src = 'C:/Images/CompanyLogo.jpg' />
<span style="font-size:1.13em;color:#003366;">Computer Motherboard Test</span>
```

The HTML `<img>` tag provides additional functionality, such as aligning the image to the left, right, or center. Refer to the W3Schools website, located at www.w3schools.com, for more information about the `<img>` tag.

3. Search for the `ADD_FOOTER_INFO` comment and uncomment the following lines of code before the closing `</body>` tag to add the text `TestStand Generated Report` to the footer of the report:

```
<span style="font-size:1.13em;color:#003366;">TestStand Generated Report</span>
```

4. Add `CompanyLogo.jpg` to the `C:\Images` directory.

TR6_Horizontal.xsl or TR5_Horizontal.xsl

The following figure shows a modified report header using the `TR6_Horizontal.xsl` or `TR5_Horizontal.xsl` file.



UUT REPORT

STATION ID	INRD-VS
SERIAL NUMBER	NONE
DATE	Monday, September 23, 2024
TIME	6:14:59 PM
OPERATOR	administrator
EXECUTION TIME	1.86744 seconds
NUMBER OF RESULTS	17
UUT RESULT	Passed

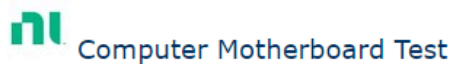
The following figure shows a modified report footer using the `TR6_Horizontal.xsl` or `TR5_Horizontal.xsl` file.

END UUT REPORT

TestStand Generated Report

TR6_report.xml or TR5_report.xml

The following figure shows a modified report header using the TR6_report.xml or TR5_report.xml file.



UUT REPORT

STATION ID	INRD-VS
SERIAL NUMBER	NONE
DATE	Monday, September 23, 2024
TIME	6:15:27 PM
OPERATOR	administrator
EXECUTION TIME	1.81764 seconds
NUMBER OF RESULTS	17
UUT RESULT	Passed

The following figure shows a modified report footer using the TR6_report.xml or TR5_report.xml file.

END UUT REPORT

TestStand Generated Report

TR6_Expand.xml or TR5_Expand.xml

The following figure shows a modified report header using the TR6_Expand.xml or TR5_Expand.xml file.



Computer Motherboard Test



Expand/Collapse All Step Results

UUT REPORT

STATION ID	INRD-VS
SERIAL NUMBER	NONE
DATE	Monday, September 23, 2024
TIME	6:15:59 PM
OPERATOR	administrator
EXECUTION TIME	1.80616 seconds
NUMBER OF RESULTS	17
UUT RESULT	Passed

The following figure shows a modified report footer using the TR6_Expand.xml or TR5_Expand.xml file.

END UUT REPORT

TestStand Generated Report

Adding Step Results to an ATML Test Results 6.01 or 5.0 Report without Indentation

Complete the following steps to ensure a report does not indent any step results using the TR6_Horizontal.xml, TR5_Horizontal.xml, TR6_report.xml, TR5_report.xml, TR6_Expand.xml, or TR5_Expand.xml files.

1. Open the .xml file you want to change.
2. Search for the Customization variables comment.
3. Search for the gRemoveIndentationFunctionality variable and change the value of the select attribute from "false" to "true", which results in the following line of code:

```
<xsl:variable name="gRemoveIndentationFunctionality"select="true()" />
```


The following figure shows a modified report without indentation using the `TR6_Horizontal.xml` or `TR5_Horizontal.xml` file.

Begin Sequence: Select & Case Steps (C:\Customization\Flow Control Step Types.seq)						
Step	Status	Measurement	Units	Limits		
				Low Limit	High Limit	Comparison Type
Prompt for Text Response	Done					
TestResults/Data						
ButtonHit:	1					
Response:	yes					
Select(yes) RunState.Sequence.Main["Prompt for Text Response"].Result.Response	Done					
Step	Status	Measurement	Units	Limits		
				Low Limit	High Limit	Comparison Type
Case("y", "yes")	Done					

The following figure shows a modified report without indentation using the `TR6_report.xml` or `TR5_report.xml` file.

Begin Sequence: Select & Case Steps (C:\Customization\Flow Control Step Types.seq)	
Prompt for Text Response	
Status:	Done
TestResults/Data:	
ButtonHit:	1
Response:	yes
Select(yes) RunState.Sequence.Main["Prompt for Text Response"].Result.Response	
Status:	Done
Case("y", "yes")	
Status:	Done

The following figure shows a modified report without indentation using the `TR6_Expand.xml` or `TR5_Expand.xml` file.

✓  **Begin Sequence: Select & Case Steps**
(C:\Customization\Flow Control Step Types.seq)

 **Prompt for Text Response**

 **Select**

 **Case**

 **Yes Case**

 **End**

 **End**

End Sequence: Select & Case Steps

Adding Sequence Call Step Results to an ATML Test Results 6.01 or 5.0 Report without Indentation

Complete the following steps to ensure that a report does not indent Sequence Call step results using the TR6_report.xml, TR5_report.xml, TR6_Expand.xml, or TR5_Expand.xml file.

1. Open the .xml file you want to change.
2. Search for the `<!-- Customization variables-->` comment.
3. Search for the `gIndentSequenceCalls` variable and change the value of the `select` attribute from `true` to `false`, which results in the following line of code:

```
<xsl:variable name="gIndentSequenceCalls"
select="false()" />
```



Note Flow Control steps continue to indent even with this customization.

TR6_report.xsl or TR5_report.xsl

The following figure shows a modified report that includes Sequence Call steps without indentation using the TR6_report.xsl or TR5_report.xsl file.

CPU Diagnostics	
Status:	Failed
Module Time:	0.226103
Begin Sequence: CPU Diagnostics (C:\Customization\Computer Motherboard Test Sequence.seq)	
Register Diagnostics	
Status:	Passed
Module Time:	0.001783
Instruction Set Diagnostics	
Status:	Failed
Module Time:	0.003383
Cache Diagnostics	
Status:	Passed
Module Time:	0.001808
FPU Diagnostics	
Status:	Passed
Module Time:	0.002429
End Sequence: CPU Diagnostics	

TR6_Expand.xsl or TR5_Expand.xsl

The following figure shows a modified report that includes Sequence Call steps without indentation using the TR6_Expand.xsl or TR5_Expand.xsl file.

✖ CPU Test	
CPU Test	
Status:	Failed
Module Time:	2.815587
✖ Begin Sequence: CPU Test (C:\Customization\Computer Motherboard Test Sequence.seq)	
✔ Register Test	

Changing the Display Format of Execution Time in an ATML Test Results 6.01 or 5.0 Report

XSL and JavaScript code generate the HTML code for the execution time information, for which the default display is in seconds. Complete the following steps to display the execution time information in `Hour:Minutes:Seconds` format.

1. Open the `.xsl` file you want to change.
2. Search for the `CHANGE_TOTAL_TIME_FORMAT` comment.
3. Delete the following line of code: `<xsl:value-of select="tr:ResultSet/tr:Extension/ts:TSTStepProperties/ts:TotalTime/@value"/> seconds`
4. Uncomment the following lines of code:

```
<xsl:call-template name="GetTotalTimeInHHMMSSFormat">
  <xsl:with-param name="timeInSeconds" select="tr:ResultSet/tr:Extension/
ts:TSTStepProperties/ts:TotalTime/@value"/>
</xsl:call-template>
```

TR6_Horizontal.xsl or TR5_Horizontal.xsl

The following figure shows the modified execution time in a report using the `TR6_Horizontal.xsl` or `TR5_Horizontal.xsl` file.

UUT Report	
Station ID	DDT-6KBXD1
Serial Number	NONE
Date	Monday, June 09, 2014
Time	3:52:29 PM
Operator	administrator
Execution Time	00:00:03.711
Number of Results	26
UUT Result	Failed

TR6_report.xml or TR5_report.xml

The following figure shows the modified execution time in a report using the TR6_report.xml or TR5_report.xml file.

UUT Report	
Station ID:	DDT-6KBXDX1
Serial Number:	NONE
Date:	Monday, June 09, 2014
Time:	3:57:58 PM
Operator:	administrator
Execution Time:	00:00:03.125
Number of Results:	26
UUT Result:	Failed
Failure Chain:	

TR6_Expand.xml or TR5_Expand.xml

The following figure shows the modified execution time in a report using the TR6_Expand.xml or TR5_Expand.xml file.

 Expand/Collapse All Step Results	
UUT Report	
Station ID:	DDT-6KBXDX1
Serial Number:	NONE
Date:	Monday, June 09, 2014
Time:	4:02:42 PM
Operator:	administrator
Execution Time:	00:00:04.589
Number of Results:	26
UUT Result:	Failed
Failure Chain:	

Customizing the UUT Report Header for an ATML Test Results 6.01 or 5.0 Report

Complete the following steps to add a row in the header of a report to display the ID of the result set the report contains.

1. Open the `.xsl` file.
2. Search for the `CREATE_UUTHEADER_INFO` comment and uncomment the following table row code that appears after the comment:

```
<tr>
  <td>
    ResultSet ID
  </td>
  <td>
    <xsl:value-of select="tr:ResultSet/@ID"/>
  </td>
</tr>
```

The expression `"tr:ResultSet/@ID"` specifies the XPath expression to obtain the result set ID from the XML file. You must customize the XPath expression to match the data you want to add to the additional rows. Refer to the World Wide Web Consortium (W3C) website, located at www.w3.org, for more information about XPath.

Modifying Cell Background Color Based on Step Status in an ATML Test Results 6.01 or 5.0 Report

None

TR6_Horizontal.xsl or TR5_Horizontal.xsl

Complete the following steps to modify the background color of the step status cell using the `TR6_Horizontal.xsl` or `TR5_Horizontal.xsl` file.

1. Open the .xsl file you want to change.
2. Search for the ADD_STATUS comment.
3. Uncomment the following lines of code:

```
<xsl:variable name="cellBackgroundColor"
  <xsl:choose>
    <xsl:when test="$outcome = 'Passed'">#FFFF00</xsl:when>
    <xsl:otherwise><xsl:value-of
select="$statusBackgroundColor"/></xsl:otherwise>
  <xsl:choose>
    <td style="background-
color:{$cellBackgroundColor};width:{$gSecondColumnWidth};">
      The <xsl:when test="$outcome = 'Passed'">#FFFF00</xsl:when> expression
changes the background color variable to yellow. You can use a custom XPath
expression instead.
      ATML Test Results 2.02 Report Style Sheet does not need to be updated as it
already references XPath.
```

4. Comment out the following line of code: `<td style="color:{$statusColor};text-align:center;">`
5. Optionally, you can add Failed.jpg to the C:\Images directory and uncomment the following lines of code after the ADD_IMG comment to add an image next to the UUT status text for Failed steps:

```
<xsl:if test="$outcome = 'Failed'">
  <img src = "C:\Images\Failed.jpg"/>
</xsl:if>
```

The following figure shows a modified report with yellow cells for steps with a Passed status using the TR6_Horizontal.xsl or TR5_Horizontal.xsl file.

Turn Vacuum Table On	Done					
Powerup Test	Passed					

TR6_report.xsl, TR5_report.xsl, TR6_Expand.xsl, or TR5_Expand.xsl

Complete the following steps to modify the background color of the step status cell using the TR6_report.xsl, TR5_report.xsl, TR6_Expand.xsl, or TR5_Expand.xsl file.

1. Open the .xsl file you want to change.

2. Search for the `ADD_STATUS` comment.
3. Uncomment the following lines of code:

```
<xsl:variable name="cellBackgroundColor"
  <xsl:choose>
    <xsl:when test="$outcome = 'Passed'">#FFFF00</xsl:when>
    <xsl:otherwise><xsl:value-of
select="$statusBackgroundColor"/></xsl:otherwise>
  <xsl:choose>
    <td style="background-
color:{$cellBackgroundColor};width:{$gSecondColumnWidth};">
      The <xsl:when test="$outcome = 'Passed'">#FFFF00</xsl:when> expression
changes the
      background color variable to yellow. You can use a custom XPath expression
instead.
      ATML Test Results 2.02 Report Style Sheet does not need to be updated as it
already
      references XPath.
```

4. Comment out the following line of code: `<td style="background-color:{$statusBackgroundColor};width:{$gSecondColumnWidth};">`
5. Optionally, you can add `Failed.jpg` to the `C:\Images` directory and uncomment the following lines of code after the `ADD_IMG` comment to add an image next to the UUT status text for Failed steps:

```
<xsl:if test="$outcome = 'Failed'">
  <img src = "C:\Images\Failed.jpg"/>
</xsl:if>
```

The following figure shows a modified report with yellow cells for steps with a `Passed` status using the `TR6_report.xsl`, `TR5_report.xsl`, `TR6_Expand.xsl`, or `TR5_Expand.xsl` file.

Powerup Test	
Status:	Passed
Module Time:	0.0003394

Rendering HTML Elements and New Line Characters in an ATML Test Results 6.01 or 5.0 Report

Complete the following steps to ensure a report renders HTML elements and new line characters using the `TR6_Horizontal.xml`, `TR5_Horizontal.xml`, `TR6_report.xml`, `TR5_report.xml`, `TR6_Expand.xml`, or `TR5_Expand.xml` files.

1. Open the .xml file you want to change.
2. Search for the `Customization variables` comment.
3. Search for the `gDisplayNewLineAndHTMLElementInStringProperty` variable and change the value of the `select` attribute from `"false"` to `"true"`, which results in the following line of code:

```
<xsl:variable
name="gDisplayNewLineAndHTMLElementInStringProperty"select="true()"/>
```

For example, if you create a local string variable named `StringContainingHTMLElementAndNewLine` and specify `Locals.StringContainingHTMLElementAndNewLine="<b>Hello\nWorld</b>"` in the `TestStand` expression, executing the sequence file generates a modified report, as shown for each file type in the following figures:

`TR6_Horizontal.xml` or `TR5_Horizontal.xml`

Step	Status	Measurement	Units	Limits		
				Low Limit	High Limit	Comparison Type
Additional Results	Done					
TestResults/Data						
StringContainingHTMLElementAndNewLine:	Hello World					

`TR6_report.xml` or `TR5_report.xml`

Additional Results	
Status:	Done
TestResults/Data:	
StringContainingHTMLElementAndNewLine:	Hello World

TR6_Expand.xsl or TR5_Expand.xsl

 Additional Results									
<table> <tr> <th colspan="2">Additional Results</th></tr> <tr> <td>Status:</td><td>Done</td></tr> <tr> <td colspan="2">TestResults/Data:</td></tr> <tr> <td>StringContainingHTMLElementAndNewLine:</td><td>Hello World</td></tr> </table>		Additional Results		Status:	Done	TestResults/Data:		StringContainingHTMLElementAndNewLine:	Hello World
Additional Results									
Status:	Done								
TestResults/Data:									
StringContainingHTMLElementAndNewLine:	Hello World								

Customizing the ATML Test Results 2.02 Report Style Sheet

You can use a text editor to customize the `TR_horizontal.xsl` report style sheet you can use for a report that validates against version 2.02 of the ATML Test Results and Session Information schema in the following ways:

- Add columns to the report
- Enable or disable table indentation
- Display or hide TestStand-specific information
- Display the fully expanded state of the report in a plain HTML format
- Customize the UUT report header



Note Copy the existing style sheets and all support files from the `<TestStand>\Components\Models\TestStandModels\ATML\StyleSheets` directory to the `<TestStand Public>\Components\Models\TestStandModels\ATML\StyleSheets` directory and make changes to the copies of the files. When you change the name of a style sheet, use the Contents tab of the Report Options dialog box to use the new style sheet when generating ATML reports.

Related concepts:

- [ATML Report Style Sheets](#)
- [ATML Test Results Reports](#)
- [Adding Columns to an ATML Test Results 2.02 Report](#)

- [Enabling or Disabling Table Indentation in an ATML Test Results 2.02 Report](#)
- [Displaying or Hiding TestStand-Specific Information in an ATML Test Results 2.02 Report](#)
- [Displaying the Fully Expanded ATML Test Results 2.02 Report in Plain HTML](#)
- [Customizing the UUT Report Header for an ATML Test Results 2.02 Report](#)
- [TestStand Directory Structure](#)

Adding Columns to an ATML Test Results 2.02 Report

Complete the following steps to modify the `TR_horizontal.xsl` style sheet to modify the XSL templates and JavaScript code to add a new StepID column to the right side of the report table. You can also copy and add the following code for each new column you want to add.

1. Open the `TR_horizontal.xsl` file.
2. Search for the `CREATE_EXTRA_COLUMNS` comment in the `StartTable` JavaScript function. Copy the following comment, paste the copy below the original, and uncomment the copy to add a new StepID column:

```
"<td rowspan='2' valign='bottom' align='center'
width='20%'><font size='1'><b>StepID</b></font></td>" +
```

The `StepID` value is the heading of the new column. Modify this value to customize the column heading.

3. Search for the `INITIALIZE_NUMBER_OF_CUSTOM_COLUMNS` comment and increment the value of the `gNoOfCustomColumns` variable by the number of columns you want to add. Because you are adding only the StepID column in this example, replace the 0 value with 1, which results in the following line of code:


```
<xsl:variable name="gNoOfCustomColumns" select="1"/>
```
4. Search for each instance of the `ADD_COLUMN_DATA` comment and the uncomment the following code:

```
<td align="right">
  <xsl:value-of select="nl:Extension/ts:TSSStepProperties/ts:StepId"/>
</td>
```

The expression `"n1:Extension/ts:TSTestProperties/ts:StepId"` specifies the XPath expression to obtain the step ID from the XML file. You must customize the XPath expression to match the data you want to add to the additional columns. Refer to the World Wide Web Consortium (W3C) website, located at www.w3.org, for more information about XPath.

5. Repeat steps 2 through 4 to add additional columns.

Enabling or Disabling Table Indentation in an ATML Test Results 2.02 Report

Complete the following steps to enable or disable the indentation of all tables the report that uses the `TR_horizontal.xsl` report style sheet displays. By default, the report displays indentation of all tables.

1. Open the `TR_horizontal.xsl` file.
2. Search for the `gIndentTables` XSLT variable.
3. Set the variable to `false()` to disable the indentation of all tables. You can reset the variable to `true()` to enable displaying indentation.

Displaying or Hiding TestStand-Specific Information in an ATML Test Results 2.02 Report

By default, the report does not display TestStand-specific properties. You can enable the `TR_horizontal.xsl` report style sheet to display the following TestStand-specific properties in the report:

- TestStand Step ID
- Step Type
- Step Group
- Block Level
- Index

- Step Caused Sequence Failure
- Loop Index
- Number of Iterations
- Number of Steps Passed
- Number of Steps Failed
- Ending Loop Index
- Interactive Execution #
- Remote Server
- Partial TPS Executed
- Number of Results
- Test Socket Index



Note The report always displays the Block Level, Index, Interactive Execution #, and Test Socket Index TestStand-specific properties as unsigned integers.

Complete the following steps to display TestStand-specific properties in the report. When you enable the report to display TestStand-specific properties, the properties appear in a separate collapsible section in the report.

1. In TestStand, enable the **Include TestStand Extension Elements** option on the Contents tab of the Report Options dialog box.
2. Open the `TR_horizontal.xsl` file in a text editor.
3. Search for the `gShowTestStandSpecificInformation` XSLT variable.
4. Set the variable to `true()` to display TestStand-specific properties. You can reset the variable to `false()` to disable displaying TestStand-specific properties in the report.



Note The report always displays the Partial TPS Executed, Number of Results, and Test Socket Index properties even when you specify a value of `false()` for the `gShowTestStandSpecificInformation` XSLT variable.

TestStand Extension Elements

The ATML Test Results and Session Information schema defines extension elements so

users can add industry-specific or product-specific data to the report.

TestStand ATML report generators log the following additional step properties to the report in `<Extension>` child elements of the `<TestGroup>`, `<SessionAction>`, and `<Test>` elements. Enable the **Include TestStand Extension Elements** option on the Contents tab of the Report Options dialog box to include these extension elements in the generated report. Refer to the `TestResultsTSExtension.xsd` file, located in the `<TestStand>\Components\Models\TestStandModels\ATML\Schemas\ATML 5` directory and the `<TestStand>\Components\Models\TestStandModels\ATML\Schemas\ATML 2.02` directory, for more information about the elements you can add to the `<Extension>` child elements.

- TestStand Step ID
- Step Type
- Step Group
- Block Level
- Step Caused Sequence Failure
- Index
- Loop Index
- Number of Iterations
- Number of Steps Passed
- Number of Steps Failed
- Ending Loop Index
- Interactive Execution #
- Remote Server
- Partial TPS Executed
- Number of Results
- Test Socket Index

TestStand ATML report generators add the following step properties only for ATML 6.01 or 5.0 reports:

- Total time
- Module time

TestStand ATML report generators add the following TestStand-specific information in the `<Extension>` child element for the `<TestResults>` and `<TestResult>` element:

- Critical failure stack
- Number of step results
- Batch and Parallel report
 - Batch serial Number
 - Test socket Index
- TSR files
 - File name
 - File ID
 - Is TSR file closed
 - TSR file status

Related concepts:

- [TestStand Directory Structure](#)

Displaying the Fully Expanded ATML Test Results 2.02 Report in Plain HTML

Complete the following steps to display the fully expanded state of the report in a plain HTML format without any additional expand or collapse functionality. By default, the report enables the expand and collapse functionality.

1. Open the `TR_horizontal.xsl` file.
2. Search for the `gGeneratePlainHTML` XSLT variable.
3. Set the variable to `true()` to display the plain HTML format. You can reset the variable to `false()` to disable displaying the plain HTML format.

Customizing the UUT Report Header for an ATML Test Results 2.02 Report

Complete the following steps to add a row in the header of a report that uses the `TR_horizontal.xsl` report style sheet to display the ID of the result set the report contains.

1. Open the `TR_horizontal.xml` file.
2. Search for the `CREATE_UUTHEADER_INFO` comment and uncomment the following table row code that appears after the comment:

```
<tr level="0">
  <td>
    <font size="1"><b>ResultSet ID</b></font>
  </td>
  <td>
    <xsl:value-of select="n1:ResultSet/@ID"/>
  </td>
</tr>
```

The expression `"n1:ResultSet/@ID"` specifies the XPath expression to obtain the result set ID from the XML file. You must customize the XPath expression to match the data you want to add to the additional rows. Refer to the World Wide Web Consortium (W3C) website, located at www.w3.org, for more information about XPath.

Customizing the ATML Test Results Report Generator

You can customize an ATML Test Results (TR) report TestStand creates by implementing a DLL that exports one or more exported extension functions. TestStand calls the functions while creating the report.

Complete the following steps to customize an ATML TR report.

1. Create an extension DLL to implement one or more of the supported exported functions.
2. In the TestStand Sequence Editor, open or create a sequence file for which you want to generate a customized report.
3. Select **Edit»Sequence File Callbacks** to launch the Sequence File Callbacks dialog box.
4. Select the **ReportOptions** callback sequence and click **Add** to override the ReportOptions callback in the sequence file. Click **OK** to close the Sequence File Callbacks dialog box.
5. On the Sequences pane, double-click **ReportOptions** to open the ReportOptions callback sequence.
6. Insert a Statement step in the ReportOptions callback sequence.
7. With the Statement step selected, click the Expression edit tab of the Step Settings

pane.

8. Enter the following expression in the **Expression** control:

```
Parameters.ReportOptions.Atml_Rpt_Options.ExtensionDLLAbsolutePa
= <absolute path to extension DLL>
```

TestStand resolves the value of the `Parameters.ReportOptions.Atml_Rpt_Options.ExtensionDLLAbsolutePa` variable at run time, so you can implement run-time logic in the ReportOptions callback to assign different values to this variable. For example, the value of the `Parameters.ReportOptions.Atml_Rpt_Options.ExtensionDLLAbsolutePa` variable might be different depending on whether the `Parameters.ReportOptions.ReportStyle` variable is ATML or ATML5.

Related concepts:

- [ATML Test Results Reports](#)
- [Using Callback Sequences to Modify Process Models](#)

Using NI DataFinder with TestStand

Using NI DataFinder to index TestStand report files in directories to which TestStand writes report files can lead to report generation errors in TestStand because of file access contention issues. To avoid these issues, do not configure DataFinder to index such directories. Instead, copy completed report files from the location to which TestStand generates the report files to another directory you have configured DataFinder to index.

For DataFinder 2011 or later and TestStand 2010 SP1 or later, DataFinder no longer indexes the `<TestStand Public>` directory by default. If TestStand 2010 SP1 is installed on the computer, DataFinder 2011 or later no longer index the `<TestStand Public>` directory of earlier versions of TestStand installed on the same computer.

If you must generate reports in a directory indexed by NI DataFinder, DataFinder 2012 SP1 introduced a delayed indexing feature which allows you to configure DataFinder to not index files until a specified amount of time has passed since the last modification of the file. This feature is enabled for My DataFinder 2012 SP1 or newer with a default delay of 20 seconds, and disabled for DataFinder Server Edition by default. Complete the following steps to customize the delayed indexing setting:

1. Navigate to the following location in the Windows registry:
[HKEY_LOCAL_MACHINE\Software\National Instruments\Common\DataFinder\ConfigItems]
2. Set the **DeferredIndexingEnabled** key to 00000001 to enable delayed indexing, or 00000000 to disable it. This key is a DWORD value.
3. Set the **DeferredIndexingTimeInSeconds** key to the number of seconds you would like DataFinder to wait before indexing new files. This key is a DWORD value.

For versions of DataFinder earlier than 2011 and versions of TestStand earlier than 2010 SP1 installed on the computer, DataFinder indexes the <TestStand Public> directory by default. Avoid generating TestStand report files to the <TestStand Public> directory or configure DataFinder not to index the <TestStand Public> directory.

Managing Users

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

By default, the sequence editor and the user interfaces launch the Login dialog box when you launch TestStand. You can customize the default login/logout implementation.

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



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

Related concepts:

- [Modifying Front-End Callbacks](#)
- [Creating and Deleting Users Using API](#)

Privileges

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

- **Operate**—Privileges for executing sequences and terminating and aborting executions.
- **Debug**—Privileges for controlling execution flow, executing manual and interactive executions, and editing station global variables and run-time variables.
- **Develop**—Privileges for editing and saving sequence files, editing and saving workspace files, and using source code control.
- **Configure**—Privileges for editing process model files and configuring station options, users, adapters, application settings, and report, database logging, and model options.
- **Custom**—Custom privileges you define. Customize the `NI_UserCustomPrivileges` data type to add new privileges.

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

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



Note TestStand stores the privilege categories as subproperties of the

`Privileges` property. The `Privileges` property also includes a Boolean `GrantAll` subproperty. The property `Privileges.GrantAll` applies to all privilege categories. When you set the `GrantAll` property to `True`, the user or group has all privileges. You must set the `GrantAll` property to `False` to honor privilege settings within each privilege category.

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

Accessing Privilege Settings for the Current User

Call the `CurrentUserHasPrivilege` expression function to verify in an expression that the current user has a specific privilege. Use the `Engine.CurrentUserHasPrivilege` method in the TestStand API to verify the privilege in a code module. The `Engine.CurrentUserHasPrivilege` method behaves identically to the `CurrentUserHasPrivilege` expression function.

When you call the `CurrentUserHasPrivilege` method or expression function, you must specify the property name of the privilege as a string argument. You can pass any subset of the property name tree structure to `CurrentUserHasPrivilege`. For example, you can call `CurrentUserHasPrivilege` with the following expressions to determine whether the current user has the privilege to terminate an execution:

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

You can pass `"*"` as the string argument to `CurrentUserHasPrivilege` to determine whether a user is currently logged in.

Related concepts:

- [Expressions](#)

Accessing Privilege Settings for Any User

The TestStand API includes methods to access the privileges of any user or group. Use

the `Engine.GetUser` and `Engine.GetUserGroup` methods to return a `User` object, then call the `User.HasPrivilege` method, which returns `True` when the user or any group to which the user belongs has the privilege you specify by name. When you call the `User.HasPrivilege` method on a `User` object the `Engine.GetUser` method returns, the `User.HasPrivilege` method behaves identically to the `CurrentUserHasPrivilege` method or expression function.

Defining Custom Privileges

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

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

Related concepts:

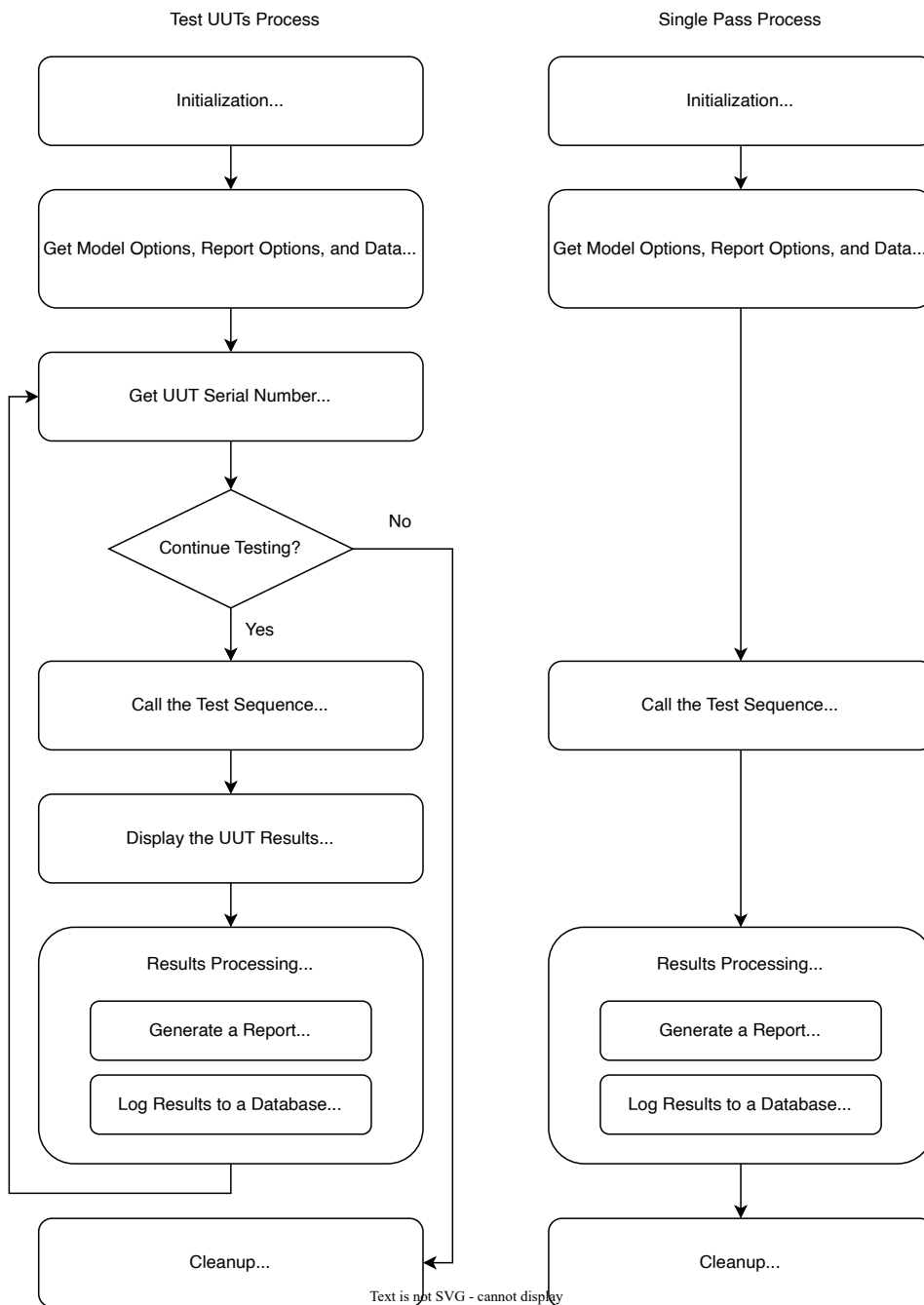
- [Modifying Types](#)
- [TestStand UI Controls](#)

Process Model Architecture

To better understand the information in this section, familiarize yourself with process models, entry points, and the relationship between a process model and a client sequence file.

The Sequential, Parallel, and Batch process models use the same basic structure for running a test sequence. Using the Test UUTs or Single Pass Execution entry point, the process models run test sequences, generate reports, and log unit under test (UUT) results to a database according to configuration settings, as shown in the following

figure.



The main differences between the process models are the number of UUTs each process model runs for the Test UUTs or Single Pass Execution entry points and the way each process model relates to and synchronizes with UUTs.



Note The TestStand Process Models Runtime package must be installed to

use the Sequential, Parallel, and Batch process models in a built TestStand Application at runtime, and the TestStand Process Models Source package must be installed to develop test sequences using the Sequential, Parallel, and Batch process models in the TestStand Sequence Editor.



Note This document reflects only the TestStand 2012 or later process models. If you are using the equivalent legacy TestStand 2010 process models, refer to the TestStand 2010 documentation for information about those process models.

Related concepts:

- [Process Models](#)
- [Entry Points](#)
- [Main Sequence and Client Sequence File](#)
- [Sequential Process Model](#)
- [Parallel Process Model](#)
- [Batch Process Model](#)
- [Database Logging](#)
- [Process Model Plug-In Architecture](#)

TestStand Process Models

The Sequential model is the default process model that you use to test one UUT at a time. Use the Parallel and Batch process models to simultaneously run the same test sequence on groups of similar UUTs. Select **Configure»Model Options** to launch the Model Options dialog box to specify the number of test sockets in the test system.

The following table lists the process models and their respective sequence files.

Process Model	Process Model Sequence File
Sequential Model	<TestStand>\Components\Models\TestStandModels\SequentialModel.seq
Parallel Model	<TestStand>\Components\Models\TestStandModels\ParallelModel.seq

Process Model	Process Model Sequence File
Batch Model	<TestStand>\Components\Models\TestStandModels\ BatchModel.seq

You can create a custom process model, or you can modify a copy of a built-in process models.

To modify the installed process model or to create a new process model, copy all the process model files from the <TestStand>\Components\Models\TestStandModels directory to <TestStand Public>\Components\Models and make changes to the copy.

The list of search paths includes the subdirectories in the <TestStand Public>\Components directory.

When you create a custom process model, use the Model tab of the Station Options dialog box to specify the custom process model sequence file as the process model for the station.



Note If you modify any process model or process model plug-in types, open and save all the process models and process model plug-ins that use the modified types to ensure you avoid a type conflict. To modify or save the process models, you must first copy all the files from the <TestStand>\Components\Models\TestStandModels directory to the <TestStand Public>\Components\Models\TestStandModels directory and modify or save the files in the <TestStand Public>\Components\Models\TestStandModels directory. The files in the <TestStand Public>\Components\Models\TestStandModels directory override the files in the <TestStand>\Components\Models\TestStandModels directory. Similarly, you must copy any process model plug-in files from the <TestStand>\Components\Models\ModelPlugins directory to the <TestStand Public>\Components\Models\ModelPlugins directory before modifying the files.

Related concepts:

- [Sequential Process Model](#)
- [Parallel Process Model](#)
- [Batch Process Model](#)
- [Copying Read-Only Files to Modify](#)
- [TestStand Directory Structure](#)
- [Search Directories](#)

Features Common to All TestStand Process Models

All TestStand process models identify UUTs and display UUT status information. Process models also call process model plug-ins that generate reports and log results to databases or files.

You can use client sequence files to customize various process model operations by overriding model-defined callback sequences. You can also create or modify process model plug-ins to add or modify functionality.

Process models provide Configuration and Execution entry points for configuring model settings and running client files under the model. Each entry point is a sequence in the process model file. The Configure and Execute menus of an application typically include the model entry points.

Related concepts:

- [Process Model Plug-In Architecture](#)
- [Database Logging](#)
- [Callback Sequences](#)
- [Creating Process Model Plug-ins](#)
- [Configuration Entry Points](#)
- [Engine Callback](#)

Configuration Entry Points

The Sequential, Parallel, and Batch process models include the following Configuration entry point sequences:

- **Configure Result Processing**—Launches the Result Processing dialog box, in which

you can enable or disable instances of the built-in and custom result processing model plug-ins and configure how the model plug-ins process test results.

- **Configure Model Options**—Launches the Model Options dialog box, in which you configure the number of test sockets and other process model-related options.

The Configuration entry points save the station report, database, and model options to the <TestStand Application Data>\Cfg directory. The report, database, and model settings apply to the test station as a whole.

Related concepts:

- [Sequential Process Model](#)
- [Parallel Process Model](#)
- [Batch Process Model](#)
- [TestStand Directory Structure](#)

Engine Callback

The Sequential, Parallel, and Batch process models include the following Engine callback:

- **ProcessModelPostResults**—The process model enables this callback when you enable the On-The-Fly option for any process model plug-in. TestStand calls this Engine callback periodically as it collects results.

Related concepts:

- [Sequential Process Model](#)
- [Parallel Process Model](#)
- [Batch Process Model](#)
- [On-the-Fly Report Generation](#)

Plug-in Utility Sequences

Process model Execution entry points call the following sequences in `ModelSupport.seq`:

- **Initialize Execution Entry Point**—Execution entry points call the Initialize

Execution Entry Point sequence at the beginning of an execution. The Initialize Execution Entry Point sequence reads the test station model options from disk and calls the ModelOptions callback so the client sequence file can modify the model options, identifies the test station name and the current user, loads the set of model plug-in instances to run during model execution, and performs other initialization operations.

- **Model Plugins – <entry point name>**—Execution entry points call the Model Plugins – <entry point name> sequences to invoke the corresponding entry points for each enabled model plug-in instance.

Related concepts:

- [Integrating Model Plug-ins with a Custom Process Model](#)

Sequential Process Model

The Sequential process model, `<TestStand>\Components\Models\TestStandModels\SequentialModel.seq`, includes sequences for the following components:

- Execution entry points
 - Test UUTs
 - Single Pass
- Configuration entry points
- Model callbacks
- Plug-in utility sequences
- Engine callbacks

Related concepts:

- [TestStand Directory Structure](#)
- [Execution Entry Points in the Sequential Process Model](#)
- [Test UUTs Execution Entry Point in the Sequential Process Model](#)
- [Single Pass Execution Entry Point in the Sequential Process Model](#)
- [Configuration Entry Points](#)
- [Model Callbacks in the Sequential Process Model](#)
- [Plug-in Utility Sequences](#)
- [Engine Callback](#)

Execution Entry Points in the Sequential Process Model

The Sequential process model includes the following Execution entry point sequences:

- **Test UUTs**—Tests and identifies multiple UUTs in a loop. The Execute menu includes the Test UUTs item when a window for a client sequence file is active.



Note When you select the Test UUTs Execution entry point to start an execution that continuously tests UUTs, any subsequent configuration changes you make to model options, report options, database logging options, or other result processing options do not affect UUTs tested in the execution.

- **Single Pass**—Tests one UUT without identifying the UUTs. The Single Pass Execution entry point performs a single iteration of the loop the Test UUTs Execution entry point performs. The Execute menu includes the Single Pass item when a window for a client sequence file is active.

Related concepts:

- [Test UUTs Execution Entry Point in the Sequential Process Model](#)
- [Single Pass Execution Entry Point in the Sequential Process Model](#)

Test UUTs Execution Entry Point in the Sequential Process Model

Open `<TestStand>\Components\Models\TestStandModels\SequentialModel.seq` in the TestStand Sequence Editor and select the **Test UUTs** sequence on the Sequences pane to examine the Sequential process model Test UUTs Execution entry point, which performs the following significant actions:

1. Calls the Initialize Execution Entry Point plug-in utility sequence.
2. Calls the Model Plugins – Begin sequence.
3. Calls the ProcessSetup callback.
4. Calls the PreUUTLoop callback.
5. Initiates the UUT For loop.
6. Calls the PreUUT callback. When no more UUTs exist, skips to .
7. Calls the Model Plugins – Pre UUT sequence. When no more UUTs exist, skips to .
8. Calls the Model Plugins – UUT Start sequence.

9. Calls the `PreMainSequence` callback.
10. Calls the `MainSequence` callback.
11. Calls the `PostMainSequence` callback.
12. Calls the Model Plugins – UUT Done sequence.
13. Calls the Model Plugins – Post UUT sequence.
14. Calls the `PostUUT` callback.
15. Ends the UUT For loop block, looping back to begin the next UUT.
16. Calls the `PostUUTLoop` callback. Performs any cleanup operations related to initialization operations performed in the `PreUUT` callback here or in the `ProcessCleanup` callback when the process model exits the UUT loop.
17. Calls the `ProcessCleanup` callback.
18. Calls the Model Plugins – End sequence. Custom process model plug-ins should perform any cleanup operations related to initialization operations performed in the Model Plugin – `PreUUT` entry point here when the process model exits the UUT loop.

Related concepts:

- [TestStand Directory Structure](#)
- [Execution Entry Points in the Sequential Process Model](#)
- [Plug-in Utility Sequences](#)
- [Model Plugin – Begin](#)
- [Model Callbacks in the Sequential Process Model](#)
- [Model Plugin – Pre UUT](#)
- [Model Plugin – UUT Start](#)
- [Model Callbacks in the Parallel Process Model](#)
- [Model Plugin – UUT Done](#)
- [Model Plugin – Post UUT](#)
- [Model Plugin – End](#)

Related information:

- [Sequential Process Model Hierarchy Diagram](#)

Single Pass Execution Entry Point in the Sequential Process Model

Open `<TestStand>\Components\Models\TestStandModels\SequentialModel.seq` in the TestStand Sequence Editor and select the **Single**

Pass sequence on the Sequences pane to examine the Sequential process model Single Pass Execution entry point. The Single Pass entry point performs the same actions as the Test UUTs entry point except that Single Pass does not include a loop for testing multiple UUTs.

Related concepts:

- [TestStand Directory Structure](#)
- [Execution Entry Points in the Sequential Process Model](#)
- [Test UUTs Execution Entry Point in the Sequential Process Model](#)

Model Callbacks in the Sequential Process Model

The Sequential process model includes the following Model callbacks, which you can override with client sequence files:

- **MainSequence**—Test UUTs and Single Pass Execution entry point sequences call the `MainSequence` callback, which is empty in the process model file. The client sequence file must contain a `MainSequence` callback that performs the tests on a UUT.
- **PreMainSequence**—The Test UUTs and Single Pass sequences call the `PreMainSequence` callback before the `MainSequence` executes. The `PreMainSequence` callback in the process model file is empty. Override this in the client file to perform an action before any code in the `MainSequence` executes.
- **PostMainSequence**—The Test UUTs and Single Pass sequences call the `PostMainSequence` callback after the `MainSequence` executes. The `PostMainSequence` callback in the process model file is empty. Override this in the client file to perform an action after all code in the `MainSequence` executes but before any Model Plug-ins execute.
- **PreUUT**—Calls the `DoPreUUT` sequence, which launches the UUT Information dialog box to obtain the UUT serial number. The Test UUTs Execution entry point calls the `PreUUT` callback at the beginning of each iteration of the UUT loop. When the operator enters a UUT serial number, the `IdentifyUUT` step stores the serial number in the **UUT.SerialNumber** parameter, which is a local variable the Test UUTs sequence passes to the `PreUUT` callback. When the operator stops testing, the UUT loop terminates, and the `IdentifyUUT` step sets the **ContinueTesting** parameter to `False`. The **ContinueTesting** parameter is another local variable the

Test UUTs sequence passes to the PreUUT callback.

- **PostUUT**—Calls the DoPostUUT sequence, which displays a pass, fail, error, or terminate banner to indicate the status of the test the `MainSequence` callback in the client sequence file performs on the UUT. The Test UUTs Execution entry point calls the PostUUT callback at the end of each iteration of the UUT loop.
- **PreUUTLoop**—Before the UUT loop begins, the Test UUTs Execution entry point calls the PreUUTLoop callback, which is empty in the process model file.
- **PostUUTLoop**—After the UUT loop terminates, the Test UUTs Execution entry point calls the PostUUTLoop callback, which is empty in the process model file.
- **ReportOptions**—The process model Execution entry point sequence calls the Model Plugins – Begin sequence in `ModelSupport.seq`. Model Plugins – Begin calls the Model Plugin – Initialize sequence in each plug-in instance. Model Plugin – Initialize in the `NI_ReportGenerator.seq` plug-in calls the ReportOptions callback so the client sequence file can modify the report options. The ReportOptions callback in the process model file is empty. This callback serves the same purpose as the more general ModelPluginOptions callback, but the report generator plug-in calls ReportOptions only for report generator plug-in instances and passes to it only reporting options.
- **DatabaseOptions**—The process model Execution entry point sequence calls the Model Plugins – Begin sequence in `ModelSupport.seq`. Model Plugins – Begin calls the Model Plugin – Initialize sequence in each plug-in instance. Model Plugin – Initialize in the `NI_DatabaseLogger.seq` plug-in calls the DatabaseOptions callback so the client sequence file can modify the database options. The DatabaseOptions callback in the process model file is empty. This callback serves the same purpose as the more general ModelPluginOptions callback, but the database logging plug-in calls DatabaseOptions only for database logging plug-in instances and passes to it only database options.
- **ModelOptions**—The process model Execution entry point sequence calls the Initialize Execution entry point sequence in `ModelSupport.seq`. After reading the test station model options from disk, the sequence calls the ModelOptions callback so the client sequence file can modify the model options. The ModelOptions callback in the process model file is empty. The client sequence file can change the value of the **ModelPluginConfigurationToLoad** parameter to specify the name of the model plug-in configuration to load from each model plug-in configuration file. The default value for the **ModelPluginConfigurationToLoad** parameter is an empty string, which directs the process model to load the active configuration from each model plug-in configuration file.

- TestReport**—The Test UUTs and Single Pass Execution entry points call the Model Plugins – UUT Done sequence in `ModelSupport.seq`. `ModelSupport.seq` calls the Model Plugin – UUT Done entry point for each configured result processor. The `NI_ReportGenerator.seq` plug-in calls the `TestReport` callback from the UUT Done entry point to generate the contents of the report for one UUT. The report generator does not call the `TestReport` callback when you enable the On-The-Fly Reporting option in the Report Options dialog box. The report plug-in defines a report for a single UUT as a header, an entry for each step result, and a footer. When you do not want to override the entire `TestReport` callback, you can override the `ModifyReportHeader`, `ModifyReportEntry`, and `ModifyReportFooter` callbacks instead to customize the report. You can override `ModifyReportEntry` only when you select the Sequence option in the Report Options dialog box and the report format is HTML or ASCII text. The `TestReport` callback returns the report using the **Report** string parameter or the **ReportSection** object reference parameter. TestStand 2010 SP1 or earlier process models use the **Report** parameter. TestStand 2012 or later process models use the **ReportSection** parameter. If you override the `TestReport` callback, you must support the parameter that corresponds to the version of the process models you use. To support both methods of returning a report, use the **ReportSection** parameter to determine which parameter the invoking process model requires. If the **ReportSection** reference uses `Nothing` as the value, you must return the report as a string in the **Report** parameter. If the **ReportSection** reference uses a value other than `Nothing`, you must return the report by updating the **ReportSection** parameter. For HTML and ASCII text report formats, depending on the settings in the Report Options dialog box, the `TestReport` callback determines whether TestStand uses sequences or a DLL to build the report body. Select the Sequence option to more easily modify the reports TestStand generates. Select the DLL option to generate reports more efficiently. When you select the Sequence option in the Report Options dialog box, the `TestReport` callback calls the `AddReportBody` sequence in `reportgen_html.seq` or `reportgen_txt.seq` to build the report body. The report generator plug-in uses a series of sequences with steps that recursively process the result list for the execution. When you select the DLL option in the Report Options dialog box, the `TestReport` callback calls a function in `modelsupport2.dll` to build the report body. You can access the project and source code for the DLL built in LabWindows/CVI from the `<TestStand>\Components\Models\TestStandModels` directory. For XML reports, `TestReport` callback calls the `AddReportBody` sequence in

`reportgen_xml.seq`. The report generator plug-in calls the `PropertyObject.GetXML` method. For ATML 2 reports, the `TestReport` callback calls the `GetATMLReport` sequence in `reportgen_atml.seq`. The report generator plug-in in calls the `Get_Atml_Report` function in `ATMLSupport.dll` in the LabWindows/CVI project. For ATML 5 reports, the `TestReport` callback calls the `AddReportBody` sequence in `reportgen_atml.seq`. The report generator plug-in calls the `Get_Atml_ReportSection_For_Body` function in `ATMLSupport.dll`. You can access the project and source code for the DLL built in LabWindows/CVI from the `<TestStand>\Components\Models\TestStandModels\ATML` directory.

- **ModifyReportHeader**—The `TestReport` callback calls the `ModifyReportHeader` callback so the client sequence file can modify the report header. The `ModifyReportHeader` callback receives parameters for the UUT information, the tentative report header text, and the report options. The `ModifyReportHeader` callback in the process model file is empty.
- **ModifyReportEntry**—The `TestReport` callback uses subsequences to call the `ModifyReportEntry` callback for each result in the result list for the UUT so the client sequence file can modify the entry point for each step result. The `ModifyReportEntry` callback receives parameters for an entry from the result list, the UUT information, the tentative report entry text, the report options, and a number that indicates the call stack depth at the time the step executed. `TestStand` calls `ModifyReportEntry` callbacks when you generate HTML or ASCII text reports and enabled the `Sequence` option in the `Report Options` dialog box. In all other cases, you must modify `modelsupport2.dll`, located in the `<TestStand>\Components\Models\TestStandModels` directory, to modify how step results appear in the report. The `ModifyReportEntry` callback in the process model file is empty.
- **ModifyReportFooter**—The `TestReport` callback calls the `ModifyReportFooter` callback so the client sequence file can modify the report footer. The `ModifyReportFooter` callback receives parameters for the UUT information, the tentative report footer text, and the report options. The `ModifyReportFooter` callback in the process model file is empty.
- **GetReportFilePath**—The report generator process model plug-in calls the `GetReportFilePath` callback so the client sequence file can inspect the report file path.
- **LogToDatabase**—Execution entry points call the `Model Plugins – UUT Done` sequence in `ModelSupport.seq`. `ModelSupport.seq` calls the `Model Plugin`

- UUT Done entry point for each configured model plug-in. The `NI_DatabaseLogger.seq` plug-in calls the `LogToDatabase` callback from the UUT Done entry point to populate a database with the results for one UUT. The database logger does not call the `LogToDatabase` callback when you enable the Use On-The-Fly Logging option on the Logging Options tab of the Database Options dialog box. The `LogToDatabase` callback receives parameters for the UUT information, the result list for the UUT, and the database options.
- **ProcessSetup**—The Test UUTs and Single Pass Execution entry points call the `ProcessSetup` callback so the client sequence file can execute any setup steps that must run only once during the execution of the process model. The `Process Setup` callback in the process model file is empty.
- **ProcessCleanup**—The Test UUTs and Single Pass Execution entry points call the `ProcessCleanup` callback from the Cleanup step group so the client sequence file can execute any cleanup steps that must run only once during the execution of the process model. The `Process Cleanup` callback in the process model file is empty.
- **ModelPluginOptions**—Execution entry points call the Model Plugins – Begin sequence in `ModelSupport.seq`, which calls the Initialize Model Plugins sequence. Initialize Model Plugins calls the `ModelPluginOptions` callback for each model plug-in instance so the client file can modify the model plug-in settings.
- **ModelPluginConfiguration**—Execution entry points call the Model Plugins – Begin sequence in `ModelSupport.seq`, which calls the Initialize Model Plugins sequence. Initialize Model Plugins calls the `ModelPluginConfiguration` callback so the client file can inspect or modify the set of model plug-ins the process model calls at run time.

Related concepts:

- [Test UUTs Execution Entry Point in the Sequential Process Model](#)
- [Single Pass Execution Entry Point in the Sequential Process Model](#)
- [Execution Entry Points in the Sequential Process Model](#)
- [Model Callbacks in the Batch Process Model](#)
- [Model Plugin – Begin](#)
- [Model Plugin – Initialize](#)
- [Model Plug in Client-File Callbacks](#)
- [Model Plugin – UUT Done](#)
- [On-the-Fly Report Generation](#)
- [TestStand Directory Structure](#)

- [On-the-Fly Database Logging](#)

Parallel Process Model

The Parallel process model, `<TestStand>\Components\Models\TestStandModels\ParallelModel.seq`, includes sequences for the following components:

- Main Execution entry points
 - Test UUTs
 - Single Pass
- Hidden Execution entry points
 - Test UUTs - Test Socket Entry Point
 - Single Pass - Test Socket Entry Point
- Configuration entry points
- Plug-in utility sequences
- Model callbacks
- Engine callbacks

Use the Parallel model to control multiple independent test sockets. With the Parallel model, you can start and stop testing on any test socket at any time. For example, if you have five test sockets for testing radios, you can load a new radio into an open test socket while the other test sockets continue testing other radios.

When you select the Single Pass Execution entry point, the Parallel model launches a separate execution for each test socket without prompting for UUT serial numbers.

Related concepts:

- [TestStand Directory Structure](#)
- [Main Execution Entry Points in the Parallel Process Model](#)
- [Test UUTs Execution Entry Point in the Parallel Process Model](#)
- [Single Pass Execution Entry Point in the Parallel Process Model](#)
- [Hidden Execution Entry Points in the Parallel Process Model](#)
- [Test UUTs – Test Socket Entry Point Execution Entry Point in the Parallel Process Model](#)
- [Single Pass - Test Socket Execution Entry Point in the Parallel Process Model](#)
- [Configuration Entry Points](#)

- [Plug-in Utility Sequences](#)
- [Model Callbacks in the Parallel Process Model](#)
- [Engine Callback](#)
- [Overriding PreUUT and PostUUT Parallel Model Callbacks](#)

Main Execution Entry Points in the Parallel Process Model

The Parallel process model includes the following main Execution entry point sequences:

- **Test UUTs**—Initiates a hidden execution that controls the test socket executions it creates using the Test UUTs - Test Socket Entry Point sequence. The Execute menu includes the Test UUTs item when a window for a client sequence file is active.
- **Single Pass**—Initiates a hidden execution that controls the test socket executions it creates using the Single Pass - Test Socket Entry Point sequence. The Execute menu includes the Single Pass item when a window for a client sequence file is active.

Related concepts:

- [Test UUTs Execution Entry Point in the Parallel Process Model](#)
- [Test UUTs – Test Socket Entry Point Execution Entry Point in the Parallel Process Model](#)
- [Single Pass Execution Entry Point in the Parallel Process Model](#)
- [Single Pass - Test Socket Execution Entry Point in the Parallel Process Model](#)
- [Hidden Execution Entry Points in the Parallel Process Model](#)

Test UUTs Execution Entry Point in the Parallel Process Model

The Test UUTs main Execution entry point is the sequence the controlling execution runs.

Open <TestStand>\Components\Models\TestStandModels\ParallelModel.seq in the TestStand Sequence Editor and select the **Test UUTs** sequence on the Sequences pane to examine the Parallel process model Test UUTs Execution entry point, which performs the following significant actions:

1. Calls the Initialize Execution Entry Point plug-in utility sequence.

2. Calls the Model Plugins – Begin sequence.
3. Calls the ProcessSetup callback.
4. Calls the Run UUT Info Dialog utility sequence.
5. Creates and initializes the test socket executions.
6. Calls the ProcessDialogRequests utility sequence.
7. Waits for test socket executions to complete.
8. Calls the ProcessCleanup callback.
9. Calls the Model Plugins – End sequence.

Related concepts:

- [Main Execution Entry Points in the Parallel Process Model](#)
- [TestStand Directory Structure](#)
- [Plug-in Utility Sequences](#)
- [Model Plugin – Begin](#)
- [Model Callbacks in the Parallel Process Model](#)
- [Utility Sequences in the Parallel Process Model](#)
- [Test UUTs – Test Socket Entry Point Execution Entry Point in the Parallel Process Model](#)
- [Model Plugin – End](#)

Related information:

- [Parallel Process Model Hierarchy Diagram](#)

Single Pass Execution Entry Point in the Parallel Process Model

The Single Pass main Execution entry point is the sequence the controlling execution runs.

Open <TestStand>\Components\Models\TestStandModels\ParallelModel.seq in the TestStand Sequence Editor and select the **Single Pass** sequence on the Sequences pane to examine the Parallel process model Single Pass Execution entry point, which performs the following significant actions:

1. Calls the Initialize Execution Entry Point plug-in utility sequence.
2. Calls the Model Plugins – Begin sequence.
3. Calls the ProcessSetup callback.

4. Calls the Run UUT Info Dialog utility sequence.
5. Creates and initializes the test socket executions.
6. Waits for test socket executions to complete.
7. Calls the ProcessCleanup callback.
8. Calls the Model Plugins – End sequence.

Related concepts:

- [Main Execution Entry Points in the Parallel Process Model](#)
- [TestStand Directory Structure](#)
- [Plug-in Utility Sequences](#)
- [Model Plugin – Begin](#)
- [Model Callbacks in the Parallel Process Model](#)
- [Utility Sequences in the Parallel Process Model](#)
- [Single Pass - Test Socket Execution Entry Point in the Parallel Process Model](#)
- [Model Plugin – End](#)

Hidden Execution Entry Points in the Parallel Process Model

The main Execution entry points of the Parallel process model use but do not display the following hidden Execution entry point sequences to initiate test socket executions:

- Test UUTs - Test Socket Entry Point—The controlling execution uses this entry point to create the test socket executions and implement the Test UUTs Execution entry point for the test socket executions.
- Single Pass - Test Socket Entry Point—The controlling execution uses this entry point to create the test socket executions and implement the Single Pass Execution entry point for test socket executions.

Related concepts:

- [Main Execution Entry Points in the Parallel Process Model](#)
- [Test UUTs – Test Socket Entry Point Execution Entry Point in the Parallel Process Model](#)
- [Test UUTs Execution Entry Point in the Parallel Process Model](#)
- [Single Pass - Test Socket Execution Entry Point in the Parallel Process Model](#)
- [Single Pass Execution Entry Point in the Parallel Process Model](#)

Test UUTs – Test Socket Entry Point Execution Entry Point in the Parallel Process Model

The Test UUTs – Test Socket Entry Point is the Execution entry point sequence the test socket executions run. The controlling execution creates the test socket executions in the Test UUTs Execution entry point sequence.

Open `<TestStand>\Components\Models\TestStandModels\ParallelModel.seq` in the TestStand Sequence Editor and select the **Test UUTs – Test Socket Entry Point** sequence on the Sequences pane to examine the Parallel process model Test UUTs – Test Socket Entry Point Execution entry point, which performs the following significant actions:

1. Calls the Initialize Execution Entry Point plug-in utility sequence.
2. Calls the Model Plugins – Begin sequence.
3. Calls the PreUUTLoop callback.
4. Initiates the UUT For loop.
5. Calls the PreUUT callback. When no more UUTs exist, skips to .
6. Calls the Model Plugins – Pre UUT sequence. When no more UUTs exist, skips to .
7. Calls the Model Plugins – UUT Start sequence.
8. Calls the `PreMainSequence` callback.
9. Calls the `MainSequence` callback.
10. Calls the `PostMainSequence` callback.
11. Calls the Model Plugins – UUT Done sequence.
12. Calls the Model Plugins – Post UUT sequence.
13. Calls the PostUUT callback.
14. Ends the UUT For loop block, looping back to begin the next UUT.
15. Calls the PostUUTLoop callback. Performs any cleanup operations related to initialization operations performed in the PreUUT callback here when the process model exits the UUT loop.
16. Calls the Model Plugins – End sequence. Custom process model plug-ins should perform any cleanup operations related to initialization operations performed in the Model Plugin – PreUUT entry point here when the process model exits the UUT loop.

Related concepts:

- [Test UUTs Execution Entry Point in the Parallel Process Model](#)
- [Main Execution Entry Points in the Parallel Process Model](#)
- [TestStand Directory Structure](#)
- [Plug-in Utility Sequences](#)
- [Model Plugin – Begin](#)
- [Model Callbacks in the Parallel Process Model](#)
- [Model Plugin – Pre UUT](#)
- [Model Plugin – UUT Start](#)
- [Model Plugin – UUT Done](#)
- [Model Plugin – Post UUT](#)
- [Model Plugin – End](#)

Single Pass - Test Socket Execution Entry Point in the Parallel Process Model

The Single Pass – Test Socket Entry Point is the Execution entry point sequence the test socket executions run. The controlling execution creates the test socket executions in the Single Pass Execution entry point sequence.

Open <TestStand>\Components\Models\TestStandModels\ParallelModel.seq in the TestStand Sequence Editor and select the **Single Pass - Test Socket Entry Point** sequence on the Sequences pane to examine the Parallel process model Single Pass - Test Socket Entry Point Execution entry point, which performs the following significant actions:

1. Calls the Initialize Execution Entry Point plug-in utility sequence.
2. Calls the Model Plugins – Begin sequence.
3. Calls the PreUUTLoop callback.
4. Calls the PreUUT callback. When no more UUTs exist, skips to .
5. Calls the Model Plugins – Pre UUT sequence. When no more UUTs exist, skips to .
6. Calls the Model Plugins – UUT Start sequence.
7. Calls the PreMainSequence callback.
8. Calls the MainSequence callback.
9. Calls the PostMainSequence callback.
10. Calls the Model Plugins – UUT Done sequence.
11. Calls the Model Plugins – Post UUT sequence.
12. Calls the PostUUT callback.
13. Calls the PostUUTLoop callback. Performs any cleanup operations related to

initialization operations performed in the PreUUT callback here when no UUTs exist and the process model steps through .

14. Calls the Model Plugins – End sequence. Custom process model plug-ins should perform any cleanup operations related to initialization operations performed in the Model Plugin – PreUUT entry point here when no UUTs exist and the process model steps through .

Related concepts:

- [Single Pass Execution Entry Point in the Parallel Process Model](#)
- [Main Execution Entry Points in the Parallel Process Model](#)
- [TestStand Directory Structure](#)
- [Plug-in Utility Sequences](#)
- [Model Plugin – Begin](#)
- [Model Callbacks in the Parallel Process Model](#)
- [Model Plugin – Pre UUT](#)
- [Model Plugin – UUT Start](#)
- [Model Plugin – UUT Done](#)
- [Model Plugin – Post UUT](#)
- [Model Plugin – End](#)

Utility Sequences in the Parallel Process Model

The Parallel process model sequences call the plug-in utility sequences, and the main Execution entry points in the Parallel process model use the following additional utility sequences:

- **Initialize TestSocket**—The controlling execution calls the Initialize TestSocket sequence to initialize the data for and create the test socket executions.
- **Tile Execution Windows**—The controlling execution calls the Tile Execution Windows sequence to tile the test socket Execution windows by building a list of executions and posting a UIMessage to the user interface that requests window tiling.
- **Monitor Threads**—The ProcessDialogRequests sequence calls the Monitor Threads sequence periodically from the controlling execution to determine whether any of the test socket executions terminated or aborted. The Monitor Threads sequence updates the ModelData for terminated or aborted test sockets to indicate the new state and updates the UUT Information dialog box the controlling execution

launches.

- **ProcessDialogRequests**—The controlling execution calls the ProcessDialogRequests sequence from the Test UUTs sequence after displaying the UUT Information dialog box, which enqueues requests for sequence names into the **ModelData.DialogRequestQueue** parameter. The ProcessDialogRequests sequence loops while waiting for those requests. When the ProcessDialogRequests sequence receives a request, it calls the requested sequence. Additionally, the ProcessDialogRequests sequence periodically calls the Monitor Threads sequence to verify the state of and update the information for the test socket executions.
- **Run UUT Info Dialog**—The controlling execution calls the Run UUT Info Dialog sequence from a new thread in the Test UUTs Execution entry point to initialize and launches the UUT Information dialog box the Test UUTs Execution entry point uses to display information and gather serial numbers for the test socket executions.
- **Continue TestSocket**—The ProcessDialogRequests sequence calls the Continue TestSocket callback to notify the test socket to continue executing. The test socket execution waits on the notification in the default implementation of the PreUUT and PostUUT callbacks.
- **Terminate TestSocket**—The ProcessDialogRequests sequence calls this dialog box request callback. The Terminate TestSocket sequence terminates the execution for the test socket the request specifies.
- **Abort TestSocket**—The ProcessDialogRequests sequence calls this dialog box request callback. The Abort TestSocket sequence aborts the execution for the test socket the request specifies.
- **Restart TestSocket**—The ProcessDialogRequests sequence calls this dialog box request callback. The Restart TestSocket sequence restarts the execution for the test socket the request specifies and re-tiles the Execution windows to include the Execution window the Restart TestSocket sequence restarts.
- **Terminate All TestSockets**—The ProcessDialogRequests sequence calls this dialog box request callback. The Terminate All TestSockets sequence terminates all the test socket executions.
- **Abort All TestSockets**—The ProcessDialogRequests sequence calls this dialog box request callback. The Abort All TestSockets sequence aborts all the test socket executions.
- **Stop All TestSockets**—The ProcessDialogRequests sequence calls this dialog box request callback. The Stop All TestSockets sequence sets a flag for each test socket execution to stop after completing the current UUT test sequence. The sequence

also sets a notification for test socket executions to continue to that point without interruption.

- **View TestSocket Report**—The `ProcessDialogRequests` sequence calls this dialog box request callback to enable the **View Report** button in the UUT Information dialog box. Click the **View Report** button to launch a report viewer for the report file for the test socket the request specifies.
- **View TestSocket Report – Current Only**—The `ProcessDialogRequests` sequence calls this dialog box request callback to enable the **View Report** button in the UUT Information dialog box. Click the **View Report** button to launch a report viewer for the last report generated for the test socket the request specifies. This sequence differs from the `View TestSocket Report` sequence in that it shows only the last report instead of the whole report file.

Related concepts:

- [Plug-in Utility Sequences](#)
- [Test UUTs Execution Entry Point in the Parallel Process Model](#)
- [Model Callbacks in the Parallel Process Model](#)

Model Callbacks in the Parallel Process Model

The Parallel process model includes the following Model callbacks, which you can override with client sequence files:

- **MainSequence**—The Test UUTs - Test Socket Entry Point and Single Pass - Test Socket Entry Point sequences call the `MainSequence` callback, which is empty in the process model file. The client sequence file must contain a `MainSequence` callback that performs the tests on a UUT.
- **PreMainSequence**—The Test UUTs and Single Pass sequences call the `PreMainSequence` callback before the `MainSequence` executes. The `PreMainSequence` callback in the process model file is empty. Override this in the client file to perform an action before any code in the `MainSequence` executes.
- **PostMainSequence**—The Test UUTs and Single Pass sequences call the `PostMainSequence` callback after the `MainSequence` executes. The `PostMainSequence` callback in the process model file is empty. Override this in the client file to perform an action after all code in the `MainSequence` executes but before any Model Plug-ins execute.

- **PreUUT**—Calls the DoPreUUT sequence, which launches the UUT Information dialog box to obtain the UUT serial numbers for the test sockets. The Test UUTs — Test Socket Entry Point sequence calls the PreUUT callback at the beginning of each iteration of the UUT loop. When the operator enters a serial number, the code for the dialog box stores the serial number in the **TestSocket.UUT.SerialNumber** parameter. When the operator stops testing, the UUT loop terminates, and the code for the dialog box sets the **TestSocket.ContinueTesting** parameter to `False`.
- **PostUUT**—Calls the DoPostUUT sequence, which displays a pass, fail, error, or terminate banner to indicate the status of the test the `MainSequence` callback in the client sequence file performs on the UUT. The Test UUTs — Test Socket Entry Point sequence calls the PostUUT callback at the end of each iteration of the UUT loop.
- **PreUUTLoop**—Before the UUT loop begins, the Test UUTs — Test Socket Entry Point sequence calls the PreUUTLoop callback, which is empty in the process model file.
- **PostUUTLoop**—After the UUT loop terminates, the Test UUTs — Test Socket Entry Point sequence calls the PostUUTLoop callback, which is empty in the process model file.
- **ReportOptions**—The process model Execution entry point sequence calls the Model Plugins – Begin sequence in `ModelSupport.seq`. Model Plugins – Begin calls the Model Plugin – Initialize sequence in each plug-in instance. Model Plugin – Initialize in the `NI_ReportGenerator.seq` plug-in calls the ReportOptions callback so the client sequence file can modify the report options. The ReportOptions callback in the process model file is empty. This callback serves the same purpose as the more general ModelPluginOptions callback, but the report generator plug-in calls ReportOptions only for report generator plug-in instances and passes to it only reporting options.
- **DatabaseOptions**—The process model Execution entry point sequence calls the Model Plugins – Begin sequence in `ModelSupport.seq`. Model Plugins – Begin calls the Model Plugin – Initialize sequence in each plug-in instance. Model Plugin – Initialize in the `NI_DatabaseLogger.seq` plug-in calls the DatabaseOptions callback so the client sequence file can modify the database options. The DatabaseOptions callback in the process model file is empty. This callback serves the same purpose as the more general ModelPluginOptions callback, but the database logging plug-in calls DatabaseOptions only for database logging plug-in instances and passes to it only database options.
- **ModelOptions**—The process model Execution entry point sequence calls the

Initialize Execution entry point sequence in `ModelSupport.seq`. After reading the test station model options from disk, the sequence calls the `ModelOptions` callback so the client sequence file can modify the model options. The `ModelOptions` callback in the process model file is empty. The client sequence file can change the value of the **ModelPluginConfigurationToLoad** parameter to specify the name of the model plug-in configuration to load from each model plug-in configuration file. The default value for the **ModelPluginConfigurationToLoad** parameter is an empty string, which directs the process model to load the active configuration from each model plug-in configuration file.

- **TestReport**—The Test UUTs and Single Pass Execution entry points call the Model Plugins – UUT Done sequence in `ModelSupport.seq`. `ModelSupport.seq` calls the Model Plugin – UUT Done entry point for each configured result processor. The `NI_ReportGenerator.seq` plug-in calls the `TestReport` callback from the UUT Done entry point to generate the contents of the report for one UUT. The report generator does not call the `TestReport` callback when you enable the On-The-Fly Reporting option in the Report Options dialog box. The report plug-in defines a report for a single UUT as a header, an entry for each step result, and a footer. When you do not want to override the entire `TestReport` callback, you can override the `ModifyReportHeader`, `ModifyReportEntry`, and `ModifyReportFooter` callbacks instead to customize the report. You can override `ModifyReportEntry` only when you select the Sequence option in the Report Options dialog box and the report format is HTML or ASCII text.

The `TestReport` callback returns the report using the **Report** string parameter or the **ReportSection** object reference parameter. TestStand 2010 SP1 or earlier process models use the **Report** parameter. TestStand 2012 or later process models use the **ReportSection** parameter. If you override the `TestReport` callback, you must support the parameter that corresponds to the version of the process models you use. To support both methods of returning a report, use the **ReportSection** parameter to determine which parameter the invoking process model requires. If the **ReportSection** reference uses `Nothing` as the value, you must return the report as a string in the **Report** parameter. If the **ReportSection** reference uses a value other than `Nothing`, you must return the report by updating the **ReportSection** parameter.

For HTML and ASCII text report formats, depending on the settings in the Report Options dialog box, the `TestReport` callback determines whether TestStand uses

sequences or a DLL to build the report body. Select the Sequence option to more easily modify the reports TestStand generates. Select the DLL option to generate reports more efficiently.

When you select the Sequence option in the Report Options dialog box, the TestReport callback calls the AddReportBody sequence in `reportgen_html.seq` or `reportgen_txt.seq` to build the report body. The report generator plug-in uses a series of sequences with steps that recursively process the result list for the execution. When you select the DLL option in the Report Options dialog box, the TestReport callback calls a function in `modelsupport2.dll` to build the report body. You can access the project and source code for the DLL built in LabWindows/CVI from the `<TestStand>\Components\Models\TestStandModels` directory.

For XML reports, TestReport callback calls the AddReportBody sequence in `reportgen_xml.seq`. The report generator plug-in calls the `PropertyObject.GetXML` method. For ATML 2 reports, the TestReport callback calls the `GetATMLReport` sequence in `reportgen_atml.seq`. The report generator plug-in in calls the `Get_Atml_Report` function in `ATMLSupport.dll` in the LabWindows/CVI project. For ATML 5 reports, the TestReport callback calls the AddReportBody sequence in `reportgen_atml.seq`. The report generator plug-in calls the `Get_Atml_ReportSection_For_Body` function in `ATMLSupport.dll`. You can access the project and source code for the DLL built in LabWindows/CVI from the `<TestStand>\Components\Models\TestStandModels\ATML` directory.

- **ModifyReportHeader**—The TestReport callback calls the ModifyReportHeader callback so the client sequence file can modify the report header. The ModifyReportHeader callback receives parameters for the UUT information, the tentative report header text, and the report options. The ModifyReportHeader callback in the process model file is empty.
- **GetReportFilePath**—The report generator process model plug-in calls the GetReportFilePath callback so the client sequence file can inspect the report file path.
- **ModifyReportEntry**—The TestReport callback uses subsequences to call the ModifyReportEntry callback for each result in the result list for the UUT so the client sequence file can modify the entry point for each step result. The

ModifyReportEntry callback receives parameters for an entry from the result list, the UUT information, the tentative report entry text, the report options, and a number that indicates the call stack depth at the time the step executed. TestStand does not call ModifyReportEntry callbacks when you enabled the DLL option in the Report Options dialog box. Instead, you must modify `modelsupport2.dll`, located in the `<TestStand>\Components\Models\TestStandModels` directory, to modify how step results appear in the report. The ModifyReportEntry callback in the process model file is empty.

- **ModifyReportFooter**—The TestReport callback calls the ModifyReportFooter callback so the client sequence file can modify the report footer. The ModifyReportFooter callback receives parameters for the UUT information, the tentative report footer text, and the report options. The ModifyReportFooter callback in the process model file is empty.
- **LogToDatabase**—Execution entry points call the LogToDatabase callback to populate a database with the results for one UUT. Execution entry points do not call the LogToDatabase callback when you enabled the Use On-The-Fly Logging option on the Logging Options tab of the Database Options dialog box. The LogToDatabase callback receives parameters for the UUT information, the result list for the UUT, and the database options.
- **ProcessSetup**—The Test UUTs and Single Pass Execution entry points call the ProcessSetup callback before creating the test socket executions so the client sequence file can execute any setup steps that must run only once during the execution of the process model. Only the controlling execution calls ProcessSetup. The test socket executions do not call the ProcessSetup callback.
- **ProcessCleanup**—The Test UUTs and Single Pass Execution entry points call the ProcessCleanup callback from the Cleanup step group so the client sequence file can execute any cleanup steps that must run only once during the execution of the process model. Only the controlling execution calls ProcessCleanup. The test socket executions do not call the ProcessCleanup callback.
- **ModelPluginOptions**—Execution entry points call the Model Plugins – Begin sequence in `ModelSupport.seq`, which calls the Initialize Model Plugins sequence. Initialize Model Plugins calls the ModelPluginOptions callback for each model plug-in instance so the client file can modify the model plug-in settings.
- **ModelPluginConfiguration**—Execution entry points call the Model Plugins – Begin sequence in `ModelSupport.seq`, which calls the Initialize Model Plugins sequence. Initialize Model Plugins calls the ModelPluginConfiguration callback so the client file can inspect or modify the set of model plug-ins the process model

calls at run time.

Related concepts:

- [Test UUTs – Test Socket Entry Point Execution Entry Point in the Parallel Process Model](#)
- [Single Pass - Test Socket Execution Entry Point in the Parallel Process Model](#)
- [Model Callbacks in the Batch Process Model](#)
- [Model Plugin – Begin](#)
- [Model Plugin – Initialize](#)
- [Model Plug in Client-File Callbacks](#)
- [Model Plugin – UUT Done](#)
- [On-the-Fly Report Generation](#)
- [TestStand Directory Structure](#)
- [On-the-Fly Database Logging](#)
- [Overriding PreUUT and PostUUT Parallel Model Callbacks](#)

Batch Process Model

The Batch process model, `<TestStand>\Components\Models\TestStandModels\BatchModel.seq`, includes sequences for the following components:

- Main Execution entry points
 - Test UTs
 - Single Pass
- Hidden Execution entry points
 - Test UUTs - Test Socket Entry Point
 - Single Pass - Test Socket Entry Point
- Configuration entry points
- Model callbacks
- Plug-in utility sequences
- Engine callbacks

Use the Batch model to control a set of test sockets that test multiple UUTs as a group. For example, if you have a set of circuit boards attached to a common carrier, use the Batch model to ensure that you start and finish testing all boards at the same time.



Note The Batch model causes socket threads to wait for each other.

With the synchronization features of the Batch model, you can direct a step that applies to the batch as a whole to run only once per batch instead of once for each UUT. You can also specify whether certain steps or groups of steps cannot run on more than one UUT at a time or whether certain steps must run on all UUTs at the same time. The Batch model generates batch reports that summarize the test results for the UUTs in the batch.

When you select the Single Pass Execution entry point, the Batch model launches a separate execution for each test socket without prompting for UUT serial numbers.

Related concepts:

- [TestStand Directory Structure](#)
- [Main Execution Entry Points in the Batch Process Model](#)
- [Test UUTs Execution Entry Point in the Batch Process Model](#)
- [Single Pass Execution Entry Point in the Batch Process Model](#)
- [Hidden Execution Entry Points in the Batch Process Model](#)
- [Test UUTs – Test Socket Entry Point Execution Entry Point in the Batch Process Model](#)
- [Single Pass - Test Socket Entry Point in the Batch Process Model](#)
- [Configuration Entry Points](#)
- [Model Callbacks in the Batch Process Model](#)
- [Plug-in Utility Sequences](#)
- [Engine Callback](#)
- [Testing UUTs in Parallel - Batch Model](#)
- [Overriding PreBatch and PostBatch Model Callbacks](#)
- [Overriding PreUUT and PostUUT Batch Model Callbacks](#)

Main Execution Entry Points in the Batch Process Model

The Batch process model includes the following main Execution entry point sequences:

- Test UUTs—Initiates an execution that controls a separate execution for each test socket using the Test UUTs - Test Socket Entry Point sequence. The Test UUTs

sequence adds the main threads of those executions to a Batch Synchronization object and controls the flow of execution using an instance of the `ControllerAndSocketSynchronizationManager` class of `ManagedModelSupport.dll`, located in `<TestStand>\Components\Models\TestStandModels\ManagedModelSupport`, so all test socket executions execute the `MainSequence` of the client sequence file together as a group. After a group of UUTs executes, the report generator process model plug-in generates a batch report, loops back to run the client sequence file on the next group of UUTs, and controls the subsidiary test socket executions to keep them synchronized. The Execute menu includes the Test UUTs item when a window for a client sequence file is active.

- **Single Pass**—Initiates an execution that controls a separate execution for each test socket using the Single Pass - Test Socket Entry Point sequence. The Single Pass sequence adds the main threads of those executions to a Batch Synchronization object and controls the flow of execution using an instance of the `ControllerAndSocketSynchronizationManager` class of `ManagedModelSupport.dll`, located in `<TestStand>\Components\Models\TestStandModels\ManagedModelSupport`, so all test socket executions execute the `MainSequence` of the client sequence file together as a group. After the group of UUTs executes, the Single Pass sequence generates a batch report and waits for all subsidiary executions to complete. The Execute menu includes the Single Pass item when a window for a client sequence file is active.

Related concepts:

- [Test UUTs Execution Entry Point in the Batch Process Model](#)
- [Test UUTs – Test Socket Entry Point Execution Entry Point in the Batch Process Model](#)
- [Batch Controller and Test Socket Synchronization Architecture](#)
- [TestStand Directory Structure](#)
- [Single Pass Execution Entry Point in the Batch Process Model](#)
- [Single Pass - Test Socket Entry Point in the Batch Process Model](#)

Test UUTs Execution Entry Point in the Batch Process Model

The Test UUTs main Execution entry point is the sequence the controlling execution

runs.

Open `<TestStand>\Components\Models\TestStandModels\BatchModel.seq` in the TestStand Sequence Editor and select the **Test UUTs** sequence on the Sequences pane to examine the Batch process model Test UUTs Execution entry point, which performs the following significant actions:

1. Calls the Initialize Execution Entry Point plug-in utility sequence.
2. Calls the Initialize ModelData sequence.
3. Calls the Model Plugins – Begin sequence.
4. Calls the ProcessSetup callback.
5. Calls the PreBatchLoop callback.
6. Creates and initializes the test socket executions.
7. Calls the Run Batch Info Dialog utility sequence.
8. Calls the `WaitForTestSockets` method to wait for and monitor test socket executions until the test sockets reach the `Initialize` synchronization point.
9. Calls the Add TestSocket Threads to Batch utility sequence.
10. Calls the `AllowTestSocketsToContinue` method to notify test sockets to continue past the `Initialize` synchronization point.
11. Begins the Batch For loop.
12. Calls the `WaitForTestSockets` method to wait for and monitor test socket executions until the test sockets reach the `GetUUTSerialNumber` synchronization point.
13. Calls the PreBatch callback.
14. Calls the Model Plugins – Pre Batch sequence.
15. When no more UUTs exist, sets the `ContinueTesting` test socket data variable to `False` for all the test sockets and marks all test sockets as enabled to allow currently disabled test sockets to complete execution.
16. Calls the Add TestSocket Threads to Batch utility sequence to add enabled test socket threads to the batch specification and remove disabled test sockets from the batch specification so they do not block running threads.
17. Calls the `AllowTestSocketsToContinue` method to notify test sockets to continue past the `GetUUTSerialNumber` synchronization point. If no more UUTs exist, skips to .
18. Calls the Model Plugins – Batch Start sequence.
19. Calls the `WaitForTestSockets` method to wait for and monitor test socket executions until the test sockets reach the `ReadyToRun` synchronization point.

20. Calls the Tile Execution Windows utility sequence.
21. Calls the `AllowTestSocketsToContinue` method to notify test sockets to continue past the `ReadyToRun` synchronization point.
22. Calls the `WaitForTestSockets` method to wait for and monitor test socket executions until the test sockets display the test socket reach the `PostMainSequence` synchronization point. The test sockets do not block at this synchronization point.
23. Calls the Model Plugins – Batch Done sequence
24. Calls the `WaitForTestSockets` method to wait for and monitor test socket executions until the test sockets reach the `ReAddToBatch` synchronization point.
25. Calls the Add TestSocket Threads to Batch utility sequence to add test socket execution threads back to the batch specification.
26. Calls the `AllowTestSocketsToContinue` method to notify test sockets to continue past the `ReAddToBatch` synchronization point.
27. Calls the `WaitForTestSockets` utility method to wait for and monitor test socket executions until the test sockets reach the `AfterPostUUT` synchronization point.
28. Calls the Model Plugins – Post Batch sequence
29. Calls the `PostBatch` callback.
30. Waits for the Status dialog box. When TestStand launches the `PostBatch` callback Status dialog box, the sequence waits for you to dismiss the dialog box if you have not already done so.
31. Calls the `AllowTestSocketsToContinue` method to notify test sockets to continue past the `AfterPostUUT` synchronization point.
32. Ends the Batch For loop, looping back to begin the next batch.
33. Calls the `WaitAndCheckExecutionStateForAllTestSockets` method to determine the first UUT error, if one occurred, and whether any UUTs terminated or aborted. Waits for all test socket executions to complete.
34. Calls the `PostBatchLoop` callback. Performs any cleanup operations related to initialization operations performed in the `PreBatch` callback here or in the `ProcessCleanup` callback when the process model skips steps through .
35. Calls the `ProcessCleanup` callback.
36. Calls the Model Plugins – End sequence. Custom process model plug-ins should perform any cleanup operations related to initialization operations performed in the Model Plugin – `PreBatch` entry point here when the process model skips steps through .

Related concepts:

- [Main Execution Entry Points in the Batch Process Model](#)
- [TestStand Directory Structure](#)
- [Plug-in Utility Sequences](#)
- [Model Plugin – Begin](#)
- [Model Callbacks in the Batch Process Model](#)
- [Test UUTs – Test Socket Entry Point Execution Entry Point in the Batch Process Model](#)
- [Utility Sequences in the Batch Process Model](#)
- [Model Plugin – Pre Batch](#)
- [Model Plugin – Batch Start](#)
- [Model Plugin – Batch Done](#)
- [Model Plugin – Post Batch](#)
- [Model Plugin – End](#)

Related information:

- [Batch Process Model Hierarchy Diagram](#)

Single Pass Execution Entry Point in the Batch Process Model

The Single Pass Execution entry point is the sequence the controlling execution runs.

Open `<TestStand>\Components\Models\TestStandModels\BatchModel.seq` in the TestStand Sequence Editor and select the **Single Pass** sequence on the Sequences pane to examine the Batch process model Single Pass Execution entry point, which performs the following significant actions:

1. Calls the Initialize Execution Entry Point plug-in utility sequence.
2. Calls the Initialize ModelData sequence.
3. Calls the Model Plugins – Begin sequence.
4. Calls the ProcessSetup callback.
5. Calls the PreBatchLoop callback.
6. Creates and initializes the test socket executions.
7. Calls the `WaitForTestSockets` method to wait for and monitor test socket executions until the test sockets reach the `Initialize` synchronization point.
8. Calls the Add TestSocket Threads to Batch utility sequence.

9. Calls the `AllowTestSocketsToContinue` method to notify test sockets to continue past the `Initialize` synchronization point.
10. Calls the `WaitForTestSockets` method to wait for and monitor test socket executions until the test sockets reach the `GetUUTSerialNumber` synchronization point.
11. Calls the `PreBatch` callback.
12. Calls the Model Plugins – Pre Batch sequence.
13. When no more UUTs exist, sets the `ContinueTesting` test socket data variable to `False` for all the test sockets and marks all test sockets as enabled to allow currently disabled test sockets to complete execution.
14. Calls the `AllowTestSocketsToContinue` method to notify test sockets to continue past the `GetUUTSerialNumber` synchronization point. If no more UUTs exist, skips to .
15. Calls the Model Plugins – Batch Start sequence.
16. Calls the `WaitForTestSockets` method to wait for and monitor test socket executions until the test sockets reach the `ReadyToRun` synchronization point.
17. Calls the Tile Execution Windows utility sequence.
18. Calls the `AllowTestSocketsToContinue` method to notify test sockets to continue past the `ReadyToRun` synchronization point.
19. Calls the `WaitForTestSockets` method to wait for and monitor test socket executions until the test sockets reach the `PostMainSequence` synchronization point. The test sockets do not block at this synchronization point.
20. Calls the Model Plugins – Batch Done sequence
21. Calls the `WaitForTestSockets` method to wait for and monitor test socket executions until the test sockets reach the `ReAddToBatch` synchronization point.
22. Calls the `Add TestSocket Threads to Batch` utility sequence to add test socket execution threads back to the batch specification.
23. Calls the `AllowTestSocketsToContinue` method to notify test sockets to continue past the `ReAddToBatch` synchronization point.
24. Calls the `WaitForTestSockets` utility method to wait for and monitor test socket executions until the test sockets reach the `AfterPostUUT` synchronization point.
25. Calls the Model Plugins – Post Batch sequence.
26. Calls the `PostBatch` callback.
27. Calls the `AllowTestSocketsToContinue` method to notify test sockets to continue past the `AfterPostUUT` synchronization point.
28. Calls the `WaitAndCheckExecutionStateForAllTestSockets` method to

determine the first UUT error, if one occurred, and whether any UUTs terminated or aborted. Waits for all test socket executions to complete.

29. Calls the PostBatchLoop callback. Performs any cleanup operations related to initialization operations performed in the PreBatch callback here or in the ProcessCleanup callback when the process model skips steps through .
30. Calls the ProcessCleanup callback.
31. Calls the Model Plugins – End sequence. Custom process model plug-ins should perform any cleanup operations related to initialization operations performed in the Model Plugin – PreBatch entry point here when the process model skips steps through .

Related concepts:

- [Main Execution Entry Points in the Batch Process Model](#)
- [TestStand Directory Structure](#)
- [Plug-in Utility Sequences](#)
- [Model Plugin – Begin](#)
- [Model Callbacks in the Batch Process Model](#)
- [Single Pass - Test Socket Entry Point in the Batch Process Model](#)
- [Utility Sequences in the Batch Process Model](#)
- [Model Plugin – Pre Batch](#)
- [Model Plugin – Batch Start](#)
- [Model Plugin – Batch Done](#)
- [Model Plugin – Post Batch](#)
- [Model Plugin – End](#)

Hidden Execution Entry Points in the Batch Process Model

The main Execution entry points of the Batch process model use but do not display the following hidden Execution entry point sequences to initiate test socket executions:

- Test UUTs - Test Socket Entry Point—The controlling execution invokes this entry point in a new execution once for each test socket to create the test socket executions.
- Single Pass - Test Socket Entry Point—The controlling execution invokes this entry point in a new execution once for each test socket to create the test socket executions.

Related concepts:

- [Main Execution Entry Points in the Batch Process Model](#)
- [Test UUTs – Test Socket Entry Point Execution Entry Point in the Batch Process Model](#)
- [Single Pass - Test Socket Entry Point in the Batch Process Model](#)

Test UUTs – Test Socket Entry Point Execution Entry Point in the Batch Process Model

The Test UUTs – Test Socket Entry Point is the Execution entry point sequence the test socket executions run. The controlling execution creates the test socket executions in the Test UUTs Execution entry point sequence.

Open `<TestStand>\Components\Models\TestStandModels\BatchModel.seq` in the TestStand Sequence Editor and select the **Test UUTs – Test Socket Entry Point** sequence on the Sequences pane to examine the Batch process model Test UUTs – Test Socket entry point Execution entry point, which performs the following significant actions:

1. Calls the Initialize Execution Entry Point plug-in utility sequence.
2. Calls the `SyncWithController` method to synchronize with the controlling execution at the `Initialize` synchronization point. The method waits until the controlling execution allows the test socket to continue past the `Initialize` synchronization point.
3. Calls the Model Plugins – Begin sequence.
4. Calls the `PreUUTLoop` callback.
5. Begins the UUT For loop.
6. Calls the `SyncWithControllerAtRestart` method to synchronize with the controlling execution at the `GetUUTSerialNumber` synchronization point. The method waits until the controlling execution allows the test socket to continue past the `GetUUTSerialNumber` synchronization point. When no more UUTs exist, skips to .
7. Calls the `PreUUT` callback. When no more UUTs exist, skips to .
8. Calls the Model Plugins – Pre UUT sequence. When no more UUTs exist, skips to .
9. Calls the `PreMainSequence` callback.
10. Calls the Model Plugins – UUT Start sequence.

11. Calls the `SyncWithController` method to synchronize with the controlling execution at the `ReadyToRun` synchronization point. The method waits until the controlling execution allows the test socket to continue past the `ReadyToRun` synchronization point.
12. Calls the `MainSequence` callback.
13. Removes the test socket thread from the batch specification in case the main sequence terminates or the client sequence file does not properly handle batch synchronization. The controlling execution adds the thread back to the batch specification before continuing past the next synchronization point.
14. Calls the `SyncWithController` method to synchronize with the controlling execution at the `PostMainSequence` synchronization point. The method does not wait for the controlling execution at this synchronization point.
15. Calls the `PostMainSequence` callback.
16. Calls the Model Plugins – UUT Done sequence.
17. Calls the Model Plugins – Post UUT sequence.
18. Calls the `SyncWithController` utility method to synchronize with the controlling execution at the `ReAddToBatch` synchronization point. The method waits until the controlling execution allows the test sockets to continue past the `ReAddToBatch` synchronization point.
19. Calls the `PostUUT` callback.
20. Calls the `SyncWithController` method to synchronize with the controlling execution at the `AfterPostUUT` synchronization point. The method waits until the controlling execution allows the test socket to continue past the `AfterPostUUT` synchronization point.
21. Ends the UUT For loop block, looping back to begin the next UUT For loop iteration.
22. Calls the `PostUUTLoop` callback. Performs any cleanup operations related to initialization operations performed in the `PreUUT` callback here when the process model skips steps through .
23. Calls the Model Plugins – End sequence. Custom process model plug-ins should perform any cleanup operations related to initialization operations performed in the Model Plugin – `PreUUT` entry point here when the process model skips steps through .

Related concepts:

- [Test UUTs Execution Entry Point in the Batch Process Model](#)

- [Main Execution Entry Points in the Batch Process Model](#)
- [TestStand Directory Structure](#)
- [Plug-in Utility Sequences](#)
- [Model Plugin – Begin](#)
- [Model Callbacks in the Batch Process Model](#)
- [Model Plugin – Pre UUT](#)
- [Model Callbacks in the Parallel Process Model](#)
- [Model Plugin – UUT Start](#)
- [Model Plugin – UUT Done](#)
- [Model Plugin – Post UUT](#)
- [Model Plugin – End](#)

Single Pass - Test Socket Entry Point in the Batch Process Model

The Single Pass – Test Socket Entry Point is the Execution entry point sequence the test socket executions run. The controlling execution creates the test socket executions in the Single Pass Execution entry point sequence.

Open `<TestStand>\Components\Models\TestStandModels\BatchModel.seq` in the TestStand Sequence Editor and select the **Single Pass - Test Socket Entry Point** sequence on the Sequences pane to examine the Batch process model Single Pass - Test Socket Entry Point Execution entry point, which performs the following significant actions:

1. Calls the Initialize Execution Entry Point plug-in utility sequence.
2. Calls the `SyncWithController` method to synchronize with the controlling execution at the `Initialize` synchronization point. The method waits until the controlling execution allows the test socket to continue past the `Initialize` synchronization point.
3. Calls the Model Plugins – Begin sequence.
4. Calls the `PreUUTLoop` callback.
5. Calls the `SyncWithController` method to synchronize with the controlling execution at the `GetUUTSerialNumber` synchronization point. The method waits until the controlling execution allows the test socket to continue past the `GetUUTSerialNumber` synchronization point. When no more UUTs exist, skips to .
6. Calls the `PreUUT` callback. When no more UUTs exist, skips to .

7. Calls the Model Plugins – Pre UUT sequence. When no more UUTs exist, skips to .
8. Calls the Model Plugins – UUT Start sequence.
9. Calls the `SyncWithController` method to synchronize with the controlling execution at the `ReadyToRun` synchronization point. The method waits until the controlling execution allows the test socket to continue past the `ReadyToRun` synchronization point.
10. Calls the `PreMainSequence` callback.
11. Calls the `MainSequence` callback.
12. Calls the `PostMainSequence` callback.
13. Removes the test socket thread from the batch specification in case the `MainSequence` terminates or the client sequence file does not properly handle batch synchronization. The controlling execution adds the thread back to the batch specification before continuing past the next synchronization point.
14. Calls the `SyncWithController` method to synchronize with the controlling execution at the `PostMainSequence` synchronization point. The method waits until the controlling execution allows the test socket to continue past the `PostMainSequence` synchronization point.
15. Calls the Model Plugins – UUT Done sequence.
16. Calls the Model Plugins – Post UUT sequence.
17. Calls the `SyncWithController` method to synchronize with the controlling execution at the `ReAddToBatch` synchronization point. The method waits until the controlling execution allows the test socket to continue past the `ReAddToBatch` synchronization point.
18. Calls the `PostUUT` callback.
19. Calls the `SyncWithController` method to synchronize with the controlling execution at the `AfterPostUUT` synchronization point.
20. Calls the `PostUUTLoop` callback. Performs any cleanup operations related to initialization operations performed in the `PreUUT` callback here when the process model skips steps through .
21. Calls the Model Plugins – End sequence. Custom process model plug-ins should perform any cleanup operations related to initialization operations performed in the Model Plugin – `PreUUT` entry point here when the process model skips steps through .

Related concepts:

- [Single Pass Execution Entry Point in the Batch Process Model](#)

- [Main Execution Entry Points in the Batch Process Model](#)
- [TestStand Directory Structure](#)
- [Plug-in Utility Sequences](#)
- [Model Plugin – Begin](#)
- [Model Callbacks in the Batch Process Model](#)
- [Model Plugin – Pre UUT](#)
- [Model Plugin – UUT Start](#)
- [Model Callbacks in the Parallel Process Model](#)
- [Model Plugin – UUT Done](#)
- [Model Plugin – Post UUT](#)
- [Model Plugin – End](#)

Utility Sequences in the Batch Process Model

The Batch process model sequences call the plug-in utility sequences and the following utility sequences:

- **Restart TestSocket**—The ProcessDialogRequests sequence calls the Restart TestSocket sequence to restart the execution for the test socket the request specifies.
- **Initialize TestSocket**—The controlling execution calls the Initialize TestSocket sequence to initialize the data for and create the test socket executions.
- **Tile Execution Windows**—The controlling execution calls the Tile Execution Windows sequence to tile the test socket Execution windows by building a list of executions and posting a UIMessage to the user interface that requests window tiling. The Tile Execution Windows sequence tiles only running, non-disabled test socket executions.
- **Add TestSocket Threads to Batch**—The Test UUTs and Single Pass Execution entry points call the Add TestSocket Threads to Batch sequence from the controlling execution to add the main threads of the test socket executions to a Batch Synchronization object. The threads remove themselves from the batch in the Test UUTs - Test Socket Entry Point and the Single Pass - Test Socket Entry Point sequences after running the Main sequence of the client sequence file to clean up the state of the batch in case the sequence terminates or the client sequence file did not properly handle batch synchronization.
- **ProcessDialogRequests**—The controlling execution calls the ProcessDialogRequests sequence from the Test UUTs Execution entry point

sequence after displaying the Batch UUT Information dialog box, which enqueues requests for sequence names into the **ModelData.DialogRequestQueue** parameter. The **ProcessDialogRequests** sequence loops while waiting for those requests. When the **ProcessDialogRequests** sequence receives a request, it calls the requested sequence. Additionally, the **ProcessDialogRequests** sequence periodically calls the **MonitorBatchThreads** method of the **ControllerAndSocketSynchronizationManager** class to verify the state of and update the information for the test socket executions.

- **Run Batch Info Dialog**—The controlling execution calls the Run Batch Info Dialog sequence from a new thread in the Test UUTs Execution entry point to initialize and run the dialog box in which users enter serial numbers and view the results for a particular run of the batch.
- **View TestSocket Report**—The **ProcessDialogRequests** sequence calls this dialog box request callback to enable the **View Report** button in the UUT Information dialog box. Click the **View Report** button to launch a report viewer for the report file for the test socket the request specifies.
- **View TestSocket Report – Current Only**—The **ProcessDialogRequests** sequence calls this dialog box request callback to enable the **View Report** button in the UUT Information dialog box. Click the **View Report** button to launch a report viewer for the last report generated for the test socket the request specifies. This sequence differs from the View TestSocket Report sequence because it shows only the last report instead of the whole report file.
- **View Batch Report**—The **ProcessDialogRequests** sequence calls this dialog box request callback to enable the **View Report** button in the UUT Information dialog box. Click the **View Report** button to launch a report viewer for the batch report file.
- **View Batch Report – Current Only**—The **ProcessDialogRequests** sequence calls this dialog box request callback to enable the **View Report** button in the UUT Information dialog box. Click the **View Report** button to launch a report viewer for the last batch report generated. This sequence differs from the View Batch Report sequence in that it shows only the last report instead of the whole batch report file.

Related concepts:

- [Plug-in Utility Sequences](#)
- [Test UUTs Execution Entry Point in the Batch Process Model](#)
- [Single Pass Execution Entry Point in the Batch Process Model](#)

- [Test UUTs – Test Socket Entry Point Execution Entry Point in the Batch Process Model](#)
- [Single Pass - Test Socket Entry Point in the Batch Process Model](#)
- [Hidden Execution Entry Points in the Batch Process Model](#)
- [Main Execution Entry Points in the Batch Process Model](#)
- [Model Callbacks in the Batch Process Model](#)

Model Callbacks in the Batch Process Model

The Batch process model includes the following Model callbacks, which you can override with client sequence files:

- **MainSequence**—The Test UUTs - Test Socket Entry Point and Single Pass - Test Socket Entry Point sequences call the `MainSequence` callback, which is empty in the process model file. The client sequence file must contain a `MainSequence` callback that performs the tests on a UUT.
- **PreMainSequence**—The Test UUTs and Single Pass sequences call the `PreMainSequence` callback before the `MainSequence` executes. The `PreMainSequence` callback in the process model file is empty. Override this in the client file to perform an action before any code in the `MainSequence` executes.
- **PostMainSequence**—The Test UUTs and Single Pass sequences call the `PostMainSequence` callback after the `MainSequence` executes. The `PostMainSequence` callback in the process model file is empty. Override this in the client file to perform an action after all code in the `MainSequence` executes but before any Model Plug-ins execute.
- **PreUUT**—The test socket executions call the `PreUUT` callback, which calls the `DoPreUUT` sequence, which is empty in the process model file. When you override the `PreUUT` callback with a client sequence file to obtain the serial number for the UUT, override the `PreBatch` callback also. The `PreBatch` callback launches a dialog box to obtain the serial numbers for all the UUTs in the batch.
- **PostUUT**—The test socket executions call the `Post UUT` callback, which calls the `DoPostUUT` sequence, which is empty in the process model file. When you override the `PostUUT` callback with a client sequence file to display the result status for a UUT, override the `PostBatch` callback also. The `PostBatch` callback launches a dialog box to show the result status for all the UUTs in the batch.
- **PreUUTLoop**—Before the UUT loop begins, the Test UUTs — Test Socket Entry Point sequence calls the `PreUUTLoop` callback, which is empty in the process

model file.

- **PostUUTLoop**—After the UUT loop terminates, the Test UUTs — Test Socket Entry Point sequence calls the PostUUTLoop callback, which is empty in the process model file.
- **ReportOptions**—The process model Execution entry point sequence calls the Model Plugins – Begin sequence in `ModelSupport.seq`. Model Plugins – Begin calls the Model Plugin – Initialize sequence in each plug-in instance. Model Plugin – Initialize in the `NI_ReportGenerator.seq` plug-in calls the ReportOptions callback so the client sequence file can modify the report options. The ReportOptions callback in the process model file is empty. This callback serves the same purpose as the more general ModelPluginOptions callback, but the report generator plug-in calls ReportOptions only for report generator plug-in instances and passes to it only reporting options.
- **DatabaseOptions**—The process model Execution entry point sequence calls the Model Plugins – Begin sequence in `ModelSupport.seq`. Model Plugins – Begin calls the Model Plugin – Initialize sequence in each plug-in instance. Model Plugin – Initialize in the `NI_DatabaseLogger.seq` plug-in calls the DatabaseOptions callback so the client sequence file can modify the database options. The DatabaseOptions callback in the process model file is empty. This callback serves the same purpose as the more general ModelPluginOptions callback, but the database logging plug-in calls DatabaseOptions only for database logging plug-in instances and passes to it only database options.
- **ModelOptions**—The process model Execution entry point sequence calls the Initialize Execution entry point sequence in `ModelSupport.seq`. After reading the test station model options from disk, the sequence calls the ModelOptions callback so the client sequence file can modify the model options. The ModelOptions callback in the process model file is empty. The client sequence file can change the value of the **ModelPluginConfigurationToLoad** parameter to specify the name of the model plug-in configuration to load from each model plug-in configuration file. The default value for the **ModelPluginConfigurationToLoad** parameter is an empty string, which directs the process model to load the active configuration from each model plug-in configuration file.
- **TestReport**—The Test UUTs and Single Pass Execution entry points call the Model Plugins – UUT Done sequence in `ModelSupport.seq`. `ModelSupport.seq` calls the Model Plugin – UUT Done entry point for each configured result processor. The `NI_ReportGenerator.seq` plug-in calls the TestReport callback from the UUT Done entry point to generate the contents of the report for one UUT. The

report generator does not call the TestReport callback when you enable the On-The-Fly Reporting option in the Report Options dialog box. The report plug-in defines a report for a single UUT as a header, an entry for each step result, and a footer. When you do not want to override the entire TestReport callback, you can override the ModifyReportHeader, ModifyReportEntry, and ModifyReportFooter callbacks instead to customize the report. You can override ModifyReportEntry only when you select the Sequence option in the Report Options dialog box and the report format is HTML or ASCII text.

The TestReport callback returns the report using the **Report** string parameter or the **ReportSection** object reference parameter. TestStand 2010 SP1 or earlier process models use the **Report** parameter. TestStand 2012 or later process models use the **ReportSection** parameter. If you override the TestReport callback, you must support the parameter that corresponds to the version of the process models you use. To support both methods of returning a report, use the **ReportSection** parameter to determine which parameter the invoking process model requires. If the **ReportSection** reference uses `Nothing` as the value, you must return the report as a string in the **Report** parameter. If the **ReportSection** reference uses a value other than `Nothing`, you must return the report by updating the **ReportSection** parameter.

For HTML and ASCII text report formats, depending on the settings in the Report Options dialog box, the TestReport callback determines whether TestStand uses sequences or a DLL to build the report body. Select the Sequence option to more easily modify the reports TestStand generates. Select the DLL option to generate reports more efficiently.

When you select the Sequence option in the Report Options dialog box, the TestReport callback calls the AddReportBody sequence in `reportgen_html.seq` or `reportgen_txt.seq` to build the report body. The report generator plug-in uses a series of sequences with steps that recursively process the result list for the execution. When you select the DLL option in the Report Options dialog box, the TestReport callback calls a function in `modelsupport2.dll` to build the report body. You can access the project and source code for the DLL built in LabWindows/CVI from the `<TestStand>\Components\Models\TestStandModels` directory.

For XML reports, TestReport callback calls the AddReportBody sequence in

`reportgen_xml.seq`. The report generator plug-in calls the `PropertyObject.GetXML` method. For ATML 2 reports, the `TestReport` callback calls the `GetATMLReport` sequence in `reportgen_atml.seq`. The report generator plug-in in calls the `Get_Atml_Report` function in `ATMLSupport.dll` in the `LabWindows/CVI` project. For ATML 5 reports, the `TestReport` callback calls the `AddReportBody` sequence in `reportgen_atml.seq`. The report generator plug-in calls the `Get_Atml_ReportSection_For_Body` function in `ATMLSupport.dll`. You can access the project and source code for the DLL built in `LabWindows/CVI` from the `<TestStand>\Components\Models\TestStandModels\ATML` directory.

- **ModifyReportHeader**—The `TestReport` callback calls the `ModifyReportHeader` callback so the client sequence file can modify the report header. The `ModifyReportHeader` callback receives parameters for the UUT information, the tentative report header text, and the report options. The `ModifyReportHeader` callback in the process model file is empty.
- **ModifyReportEntry**—The `TestReport` callback uses subsequences to call the `ModifyReportEntry` callback for each result in the result list for the UUT so the client sequence file can modify the entry point for each step result. The `ModifyReportEntry` callback receives parameters for an entry from the result list, the UUT information, the tentative report entry text, the report options, and a number that indicates the call stack depth at the time the step executed. `TestStand` does not call `ModifyReportEntry` callbacks when you enabled the DLL option in the Report Options dialog box. Instead, you must modify `modelsupport2.dll`, located in the `<TestStand>\Components\Models\TestStandModels` directory, to modify how step results appear in the report. The `ModifyReportEntry` callback in the process model file is empty.
- **ModifyReportFooter**—The `TestReport` callback calls the `ModifyReportFooter` callback so the client sequence file can modify the report footer. The `ModifyReportFooter` callback receives parameters for the UUT information, the tentative report footer text, and the report options. The `ModifyReportFooter` callback in the process model file is empty.
- **GetReportFilePath**—The report generator process model plug-in calls the `GetReportFilePath` callback so the client sequence file can inspect the report file path.
- **LogToDatabase**—Execution entry points call the `LogToDatabase` callback to populate a database with the results for one UUT. Execution entry points do not

call the `LogToDatabase` callback when you enabled the Use On-The-Fly Logging option on the Logging Options tab of the Database Options dialog box. The `LogToDatabase` callback receives parameters for the UUT information, the result list for the UUT, and the database options.

- **ProcessSetup**—The Test UUTs and Single Pass Execution entry points call the `ProcessSetup` callback before creating the test socket executions so the client sequence file can execute any setup steps that must run only once during the execution of the process model. Only the controlling execution calls `ProcessSetup`. The test socket executions do not call the `ProcessSetup` callback.
- **ProcessCleanup**—The Test UUTs and Single Pass Execution entry points call the `ProcessCleanup` callback from the Cleanup step group so the client sequence file can execute any cleanup steps that must run only once during the execution of the process model. Only the controlling execution calls `ProcessCleanup`. The test socket executions do not call the `ProcessCleanup` callback.
- **ModelPluginOptions**—Execution entry points call the Model Plugins – Begin sequence in `ModelSupport.seq`, which calls the Initialize Model Plugins sequence. Initialize Model Plugins calls the `ModelPluginOptions` callback for each model plug-in instance so the client file can modify the model plug-in settings.
- **ModelPluginConfiguration**—Execution entry points call the Model Plugins – Begin sequence in `ModelSupport.seq`, which calls the Initialize Model Plugins sequence. Initialize Model Plugins calls the `ModelPluginConfiguration` callback so the client file can inspect or modify the set of model plug-ins the process model calls at run time.

The following callback sequences are unique to the Batch process model:

- **PreBatch**—Launches a dialog box to obtain the batch and UUT serial numbers.
- **PostBatch**—Displays a pass, fail, error, or terminated banner and batch and UUT reports for each test socket.
- **PreBatchLoop**—Before looping on a batch of UUTs, the process model calls the `PreBatchLoop` callback, which is empty in the process model file. Use the `PreBatchLoop` callback to perform an action before testing the batch.
- **BatchReport**—After looping on a batch of UUTs, the process model calls the `PostBatchLoop` callback, which is empty in the process model file. Use the `PostBatchLoop` callback to perform an action after testing all batches of UUTs.
- **ModifyBatchReportHeader**—The Test UUTs and Single Pass Execution entry points call the report generator process model plug-in, which calls the `BatchReport`

callback to generate the contents of the batch report for the UUTs that ran in the last batch. The Batch process model defines a batch report for a single group of UUTs as a header, an entry for each UUT result, and a footer. When you do not want to override the entire BatchReport callback, you can override the ModifyBatchReportHeader, ModifyBatchReportEntry, and ModifyBatchReportFooter callbacks instead to customize the batch report.

- **ModifyBatchReportHeader**—The BatchReport callback calls the ModifyBatchReportHeader callback so the client sequence file can modify the batch report header. The ModifyBatchReportHeader callback receives parameters for the batch serial number, the tentative report header text, and the report options. The ModifyBatchReportHeader callback in the process model file is empty.
- **ModifyBatchReportEntry**—The BatchReport callback uses subsequences to call the ModifyBatchReportEntry callback for each test socket so the client sequence file can modify the entry for each UUT result for each test socket. The ModifyBatchReportEntry callback receives parameters for the test socket data, the batch serial number, the tentative report entry text, and the report options. The ModifyBatchReportEntry callback in the process model file is empty.
- **ModifyBatchReportFooter**—The BatchReport callback calls the ModifyBatchReportFooter callback so the client sequence file can modify the batch report footer. The ModifyBatchReportFooter callback receives parameters for the tentative report footer text and the report options. The ModifyBatchReportFooter callback in the process model file is empty.

Related concepts:

- [Test UUTs – Test Socket Entry Point Execution Entry Point in the Batch Process Model](#)
- [Single Pass - Test Socket Entry Point in the Batch Process Model](#)
- [Model Callbacks in the Batch Process Model](#)
- [TestStand Directory Structure](#)
- [Model Plugin – Begin](#)
- [Model Plugin – Initialize](#)
- [Model Plug in Client-File Callbacks](#)
- [Model Plugin – UUT Done](#)
- [On-the-Fly Report Generation](#)
- [On-the-Fly Database Logging](#)
- [ATML](#)

- [Main Execution Entry Points in the Batch Process Model](#)
- [Hidden Execution Entry Points in the Batch Process Model](#)
- [Overriding PreBatch and PostBatch Model Callbacks](#)
- [Overriding PreUUT and PostUUT Batch Model Callbacks](#)
- [Report Generation Functions and Sequences](#)
- [Utility Sequences in the Batch Process Model](#)

Batch Controller and Test Socket Synchronization Architecture

The Batch process model creates a controller execution and an execution for each test socket. When performing certain operations, the batch model synchronizes the controller and test socket executions at various synchronization points to ensure exclusive access to shared resources and data.

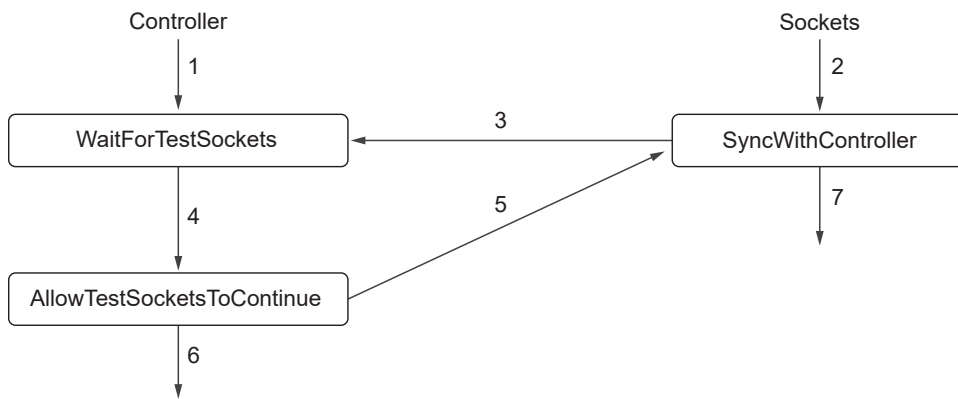
If you create a model plug-in that performs operations in the Batch Start and Batch Done controller entry points that must synchronize with operations in the UUT Start and UUT Done test socket entry points, you can call methods of the `ControllerAndSocketSynchronizationManager` class of `ManagedModelSupport.dll`, located in the `<TestStand>\Components\Models\TestStandModels\ManagedModelSupport` directory, from the controller thread or the test socket thread, as described in the following table.

Method	Called From
<code>WaitForTestSockets</code>	Controller thread
<code>SyncWithController</code>	Test socket thread
<code>AllowTestSocketsToContinue</code>	Controller thread

You can use the class to implement controller and test socket synchronization in which the sockets either to release them or to release them.

Test Sockets Wait for the Controller to Release the Sockets

The controller thread calls the `WaitForTestSockets` method, which waits until all test sockets reach a corresponding call to the `SyncWithController` method, which then waits until the controller executes a corresponding call to the `AllowTestSocketsToContinue` method, as shown in the following figure.



This synchronization implementation requires the following functionality:

- The controller passes `True` to the `socketsWaitForController` parameter of the `WaitForTestSockets` method.
- The test sockets pass `True` to the `waitForController` parameter of the `SyncWithController` method.
- The controller executes a corresponding call to the `AllowTestSocketsToContinue` method.

Test Sockets Do Not Wait for the Controller to Release the Sockets

The controller thread calls the `WaitForTestSockets` method, which waits until all test sockets reach a corresponding call to the `SyncWithController` method. The test sockets immediately continue execution after calling the `SyncWithController` method. The controller does not execute a corresponding call to the `AllowTestSocketsToContinue` method, as shown in the following figure.



This synchronization implementation requires the following functionality:

- The controller passes `False` to the `socketsWaitForController` parameter of the `WaitForTestSockets` method.

- The test sockets pass `False` to the `waitForController` parameter of the `SyncWithController` method.
- The controller must not execute a corresponding call to the `AllowTestSocketsToContinue` method.

Related concepts:

- [Batch Process Model](#)
- [Process Model Plug-In Architecture](#)
- [TestStand Directory Structure](#)

Batch Controller and Test Socket Synchronization Restrictions

The `ControllerAndSocketSynchronizationManager` class of `ManagedModelSupport.dll`, located in the `<TestStand>\Components\Models\TestStandModels\ManagedModelSupport` directory, requires a corresponding call to the `WaitForTestSockets` method for each call to the `SyncWithController` method, regardless of the controller and test socket synchronization implementation.

In addition, when you implement controller and test socket synchronization in which the sockets wait for the controller to release them, the `ControllerAndSocketSynchronizationManager` class requires a corresponding call to the `AllowTestSocketsToContinue` method for each specific call to the `WaitForTestSockets` method.

Related concepts:

- [Batch Controller and Test Socket Synchronization Architecture](#)
- [TestStand Directory Structure](#)

Implementing Batch Controller and Test Socket Synchronization for a Custom Model Plug-in

The default result processing model plug-ins perform controller and test socket synchronization when you execute a sequence using the Batch process model.

For example, the default Report result processing model plug-in, located at

`<TestStand>\Components\Models\ModelPlugins\NI_ReportGenerator.seq`, performs controller and test socket synchronization to force test sockets to serially write individual UUT reports to a file on disk after the controller writes the Batch report to a file on disk.

Use the following information to help you implement controller and test socket synchronization in a custom model plug-in.

Enabling Controller and Test Socket Synchronization for a Model Plug-in

The `NI_ModelPlugin` data type in a process model plug-in sequence file contains a `Base.RequiresBatchControllerAndSocketSynchronization` Boolean subproperty that enables or disables batch controller and test socket synchronization functionality for the plug-in. To use controller and test socket synchronization in a plug-in, ensure that this property is `True`.



Note This property is `True` by default when you use the Result Processing dialog box to create a plug-in. If you do not require controller and socket synchronization, setting this property to `False` can slightly improve performance when executing the Batch process model.

Implementing Batch Controller and Test Socket Synchronization in a Model Plug-in Entry Point

Model plug-ins use the `Locals.ControllerAndSocketSynchronizationManager` object reference to implement controller and test socket synchronization between the Model Plugin – Batch Start and Model Plugin – UUT Start entry points and between the Model Plugin – Batch Done and Model Plugin – UUT Done entry points. Model plug-ins must synchronize between these entry points because these entry points execute asynchronously to the corresponding UUT entry point counterparts. The Batch process model synchronizes all other Batch/UUT model plug-in entry point pairs.



Note

- Implementing controller and test socket synchronization in any model plug-in entry point other than the Model Plugin – Batch Start, Model Plugin – UUT Start, Model Plugin – UUT Done, and Model Plugin – Batch Done entry points might cause TestStand to hang when you execute sequences using the Batch process model.
- When you use the Result Processing dialog box to create a plug-in, the `Locals.ControllerAndSocketSynchronizationManager` object reference already exists. If you manually create a plug-in, you must correctly implement the `Locals.ControllerAndSocketSynchronizationManager` object reference. Refer to the Model Plugin – UUT Done and Model Plugin – Batch Done entry points of the default Report result processing model plug-in for an example of using the `Locals.ControllerAndSocketSynchronizationManager` object reference in an existing plug-in.

You must conditionally execute all calls to the `SyncWithController` method in the Model Plugin – UUT Start and Model Plugin – UUT Done entry points only when you execute sequences in the Batch process model. When a Model Plugin – UUT Start or Model Plugin – UUT Done entry point contains a call to the `SyncWithController` method, you can use the following precondition on the calling step to specify that the entry point executes the call only when the Batch process model calls the entry point:

```
Parameters.ModelData.ModelType == "Batch"
```

Refer to the Model Plugin – UUT Start and Model Plugin – UUT Done entry points of the default Report result processing model plug-in for an example of specifying this precondition in an existing plug-in entry point.

Related concepts:

- [Process Model Plug-In Architecture](#)
- [Batch Process Model](#)
- [TestStand Directory Structure](#)
- [Creating Process Model Plug-ins](#)

- [Model Plugin – Batch Start](#)
- [Model Plugin – UUT Start](#)
- [Model Plugin – Batch Done](#)
- [Model Plugin – UUT Done](#)
- [Model Plug-in Entry Points](#)

Report Generation Functions and Sequences

When you want to customize report generation for a test station, modify the default report generator process model plug-in or create a new report generator process model plug-in.

To modify an installed default process model plug-in, copy all files from the `<TestStand>\Components\Models\TestStandModels` directory and `<TestStand>\Components\Models\ModelPlugins\NI_ReportGenerator.seq` to the `<TestStand Public>\Components\Models\TestStandModels` and `<TestStand Public>\Components\Models\ModelPlugins` directories and make changes to the copied files. Rename the files after you modify them if you want to create a separate custom model plug-in to prevent the copied model plug-in from overwriting the default plug-in.



Note

- If you modify any process model or process model plug-in types, open and save all the process models and process model plug-ins that use the modified types to ensure you avoid a type conflict.
- If you do not rename the files and you use the files in a future version of TestStand, changes NI makes to the component might not be compatible with the modified version of the component. Storing new and customized files in the `<TestStand Public>` directory ensures that new installations of the same version of TestStand do not override the customizations and ensures that uninstalling TestStand does not remove the files you customize.

Related concepts:

- [TestStand Directory Structure](#)
- [Type Versioning](#)
- [Copying Read-Only Files to Modify](#)
- [Default Report Generator Process Model Plug-in Sequences that Generate Reports](#)
- [Default Process Model Sequences and Functions for Generating Step Results](#)
- [Model Callbacks You Can Override to Alter Report Generation](#)

Default Report Generator Process Model Plug-in Sequences that Generate Reports

The following table lists the report generator process model plug-in sequences in the `<TestStand>\Components\Models\TestStandModels` directory that generate reports.

Report Format	Header	Footer
ATML 6.01 or 5.0	AddReportHeader sequence in <code>reportgen_atml.seq</code> .	AddReportFooter sequence in <code>reportgen_atml.seq</code> .
ATML 2.02	GetATMLReport sequence in <code>reportgen_atml.seq</code> generates the header when it generates the report body.	GetATMLReport sequence in <code>reportgen_atml.seq</code> generates the footer when it generates the report body.
XML	AddReportHeader sequence in <code>reportgen_xml.seq</code> .	AddReportFooter sequence in <code>reportgen_xml.seq</code> .
HTML	AddReportHeader sequence in <code>reportgen_html.seq</code> .	AddReportFooter sequence in <code>reportgen_html.seq</code> .
Text	AddReportHeader sequence in <code>reportgen_txt.seq</code> .	AddReportFooter sequence in <code>reportgen_txt.seq</code> .

Related concepts:

- [TestStand Directory Structure](#)
- [ATML](#)
- [XML Reports](#)

Default Process Model Sequences and Functions for Generating Step Results

The following table lists the default process model sequences and C functions in the `<TestStand>\Components\Models\TestStandModels` directory that generate the report body for each step result.

Report Format Generator	Description
ATML 6.01 or 5.0	AddReportBody sequence in <code>reportgen_atml.seq</code> calls the <code>Get_Atml_ReportSection_For_Body</code> function in <code>ATML_Report.c</code> in the <code><TestStand>\Components\Models\TestStandModels\ATML\ATMLSupport.prj</code> LabWindows/CVI project.
ATML 2.02	GetATMLReport sequence in <code>reportgen_atml.seq</code> calls the <code>Get_Atml_Report</code> function in <code>ATML_Report.c</code> in the <code><TestStand>\Components\Models\TestStandModels\ATML\ATMLSupport.prj</code> LabWindows/CVI project.
XML	AddReportBody sequence in <code>reportgen_xml.seq</code> calls the TestStand API <code>PropertyObject.GetXML</code> method.
HTML sequence	AddReportBody sequence in <code>reportgen_html.seq</code> , which indirectly calls the <code>PutOneResultInReport</code> sequence for each result.
HTML DLL	<code>GetReportBody_CImplementation</code> function in <code>c_report.c</code> in the <code>modelsupport2.prj</code> LabWindows/CVI project.
Text sequence	AddReportBody sequence in <code>reportgen_txt.seq</code> , which indirectly calls the <code>PutOneResultInReport</code> sequence for each result.
Text DLL	<code>GetReportBody_CImplementation</code> function in <code>c_report.c</code> in the <code>modelsupport2.prj</code> LabWindows/CVI project.

Related concepts:

- [TestStand Directory Structure](#)
- [ATML](#)
- [XML Reports](#)

Model Callbacks You Can Override to Alter Report Generation

The following table lists the report generation Model callbacks you can override to alter the report generation for each client sequence file you run.

Section of Report to Alter	Model Callback to Override
Header	ModifyReportHeader
Footer	ModifyReportFooter
Each step result	ModifyReportEntry  Note The report generator process model plug-in calls this callback only when you generate HTML or text reports and select the Sequence option in the Select a Report Generator for Producing the Report Body section of the Contents tab of the Report Options dialog box.
Entire report	TestReport
Batch header	ModifyBatchReportHeader
Batch footer	ModifyBatchReportFooter
Each test socket result	ModifyBatchReportEntry
Entire batch report	BatchReport

Additionally, you can use the `Step.Result.ReportText` property for each step in a client sequence file to add text to the step result in the report.

Manually Selecting Process Models

1. Complete the following steps if you want to manually select a process model to use for a test station:
 - a. Select **Configure»Station Options** to launch the Station Options dialog box.
 - b. Click the **Model** tab.
 - c. Use the **Station Model** control to browse to the equivalent legacy TestStand 2010 process model you want to use located in
`<TestStand>\Components\Models\TestStandModels\`
`TestStand 2010 Process Models (Legacy).`
2. Complete the following steps if you want to manually select a process model to use for a sequence file:
 - a. Select **Edit»Sequence File Properties** to launch the Sequence File Properties dialog box.
 - b. Click the **Advanced** tab.
 - c. Select **Require Specific Model** in the **Model Option** control.
 - d. Use the **Model File** control to browse to the equivalent legacy TestStand 2010 process model you want to use.

Related concepts:

- [Search Directories](#)

Process Model Support Files

Many sequences in the TestStand process model files call functions in DLLs and subsequences in other sequence files. The following table lists the process model support files TestStand installs in the `<TestStand>\Components\Models\TestStandModels` directory.

File Name	Description
ATMLSupport.dll	DLL that contains C functions the process model sequences call to generate ATML reports.
ATMLSupport.lib	Import library for ATMLSupport.dll.
banners.c	C source for functions that display status banners.

File Name	Description
<code>BatchModel.seq</code>	Entry point and Model callbacks for the Batch process model.
<code>batchuutdlg.c</code>	C source for the functions that launch the UUT Information dialog box for the Batch process model. The <code>modelsupport2.dll</code> includes this file, but the default process model, <code>SequentialModel.seq</code> , does not call this file.
<code>c_report.c</code>	C source for generating XML, HTML, and ASCII text reports for the DLL option in the Select a Report Generator for Producing the Report Body section of the Contents tab of the Report Options dialog box. The <code>NI_ReportGenerator.seq</code> model plug-in calls this code.
<code>ColorselectPopup.c</code>	C source for the functions that launch a dialog box in which you can select a color.
<code>ColorSelectPopup.h</code>	C header file that contains declarations for the function in <code>ColorselectPopup.c</code> .
<code>main.c</code>	C source for utility functions.
<code>ManagedModelSupport.dll</code>	An assembly that contains utility methods process models call.
<code>ModelOptions.c</code>	C source for the functions that launch the Model Options dialog box and read and write the model options from and to disk.
<code>modelpanels.h</code>	C header file that contains declarations for the panels in <code>modelpanels.uir</code> .
<code>modelpanels.uir</code>	LabWindows/CVI user interface resource file that contains panels the functions in <code>modelsupport2.dll</code> use.
<code>ModelSupport.seq</code>	A sequence file that contains utility sequences process models call.
<code>modelsupport2.dll</code>	DLL that contains C functions the process model sequences call. Includes functions that launch the Report Options and Model Options dialog boxes, read and write those options from and to disk, determine the report file pathname, obtain the UUT

File Name	Description
	serial number from the operator, and display status banners. You must use the LabWindows/CVI Full Development System with the Interface to Win32 API feature installed to rebuild this DLL.
<code>modelsupport2.fp</code>	LabWindows/CVI function panels for the functions in <code>modelsupport2.dll</code> .
<code>modelsupport2.lib</code>	Import library in Microsoft Visual C/C++ format for <code>modelsupport2.dll</code> .
<code>modelsupport2.prj</code>	LabWindows/CVI project that builds <code>modelsupport2.dll</code> and <code>modelsupport2.cws</code> .
<code>NI_ResultsProcessorTemplate.seq</code>	A template sequence file the Result Processing dialog box copies when end users create a new result processing model plug-in.
<code>ParallelModel.seq</code>	Entry point and Model callbacks for the Parallel process model.
<code>paralleluutdlg.c</code>	C source for the functions that launch the UUT Information dialog box for the Parallel process model. The <code>modelsupport2.dll</code> includes this file, but the default process model, <code>SequentialModel.seq</code> , does not call this file.
<code>PropertyObject.xsd</code>	XML schema that defines the contents of the XML the <code>PropertyObject.GetXML</code> method generates and the <code>PropertyObject.SetXML</code> method requires. TestStand XML reports that <code>Report.xsd</code> defines also use <code>PropertyObject.xsd</code> .
<code>report.c</code>	C source for functions that launch the Report Options dialog box, read and write the report options from and to disk, and determine the report file pathname. The <code>NI_ReportGenerator.seq</code> model plug-in calls these sequences.
<code>report.h</code>	C header file that contains declarations for the functions in <code>report.c</code> .
<code>Report.xsd</code>	XML schema that defines the contents of TestStand XML reports.

File Name	Description
<code>reportgen_atml.seq</code>	Subsequences that add the header, result entries, and footer for a UUT to an ATML report. The <code>NI_ReportGenerator.seq</code> model plug-in calls these subsequences.
<code>reportgen_html.seq</code>	Subsequences that add the header, result entries, and footer for a UUT to an HTML report. The <code>NI_ReportGenerator.seq</code> model plug-in calls these subsequences.
<code>reportgen_txt.seq</code>	Subsequences that add the header, result entries, and footer for a UUT to an ASCII text report. The <code>NI_ReportGenerator.seq</code> model plug-in calls these subsequences.
<code>reportgen_xml.seq</code>	Subsequences that add the header, result entries, and footer for a UUT to an XML report. The <code>NI_ReportGenerator.seq</code> model plug-in calls these subsequences.
<code>ResultProcessing.c</code>	C source for functions that launch the Result Processing dialog box and read and write the process model plug-in configuration files from and to disk.
<code>ResultProcessing.h</code>	C header file that contains declarations for the functions in <code>ResultProcessing.c</code> .
<code>SequentialModel.seq</code>	Entry point and Model callbacks for the Sequential process model.
<code>uutdlg.c</code>	C source for the function that launches the UUT Information dialog box to obtain the UUT serial number from the operator.

You can view the contents of the `reportgen_atml.seq`, `reportgen_xml.seq`, `reportgen_html.seq`, and `reportgen_txt.seq` files in the TestStand Sequence Editor. These files are model sequence files and contain an empty `ModifyReportEntry` callback you can override with a client sequence file. Each `reportgen` sequence file includes a `PutOneResultInReport` sequence that calls the `ModifyReportEntry` callback.

The TestStand process model sequence files also contain an empty `ModifyReportEntry`

callback, even though no sequences in the process model sequence files call the `ModifyReportEntry` callback directly. The files contain a `ModifyReportEntry` callback so that the `ModifyReportEntry` callback appears in the Sequence File Callbacks dialog box for the client sequence file.

Related concepts:

- [TestStand Directory Structure](#)
- [ATML](#)
- [Batch Process Model](#)
- [Parallel Process Model](#)
- [Sequential Process Model](#)
- [Model Callbacks in the Sequential Process Model](#)
- [Report Generation Functions and Sequences](#)

Process Model Plug-In Architecture

To better understand the information in this section, NI recommends that you familiarize yourself with the Process Models and the Process Model Architecture topics.

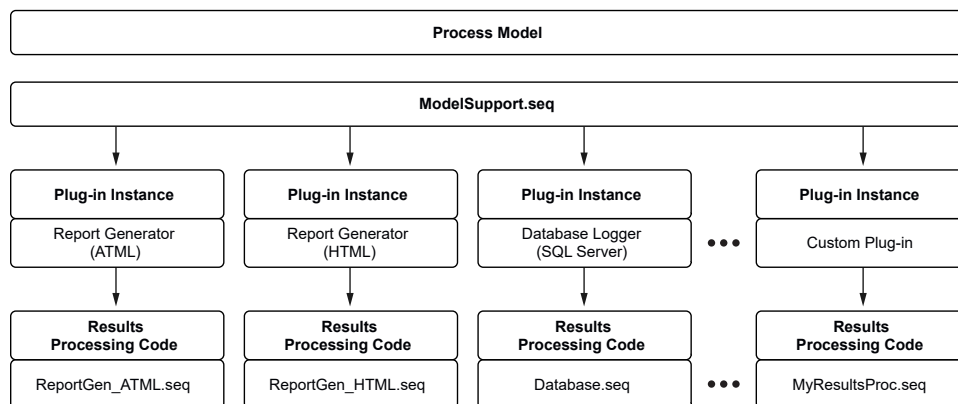
TestStand 2012 or later process models use sequence files called process model plug-ins to process test results and create result files. Each plug-in contains sequences TestStand calls during process model execution to accomplish results processing tasks, such as report generation and database logging tasks. TestStand includes built-in plug-ins to handle report generation, database logging, and raw result logging. You can create custom process model plug-ins to extend or modify the functionality of the TestStand process models without changing the TestStand process model files directly.

The TestStand 2012 or later process models include significant changes to implement a plug-in architecture for model sequence files. You can use the to achieve the following functionality:

- Decouple result processing, which includes report generation and database logging, from the process models to simplify the model files and sequence files
- More easily customize report generation and database logging functionality without having to modify the process model files and vice versa

- Output multiple reports and log to multiple databases
- Increase throughput by creating reports or logging data asynchronously to test execution
- Log result data in a fast and compact raw result format for at a time other than when sequence execution occurs or on a different computer

The following image illustrates the modular approach the TestStand 2012 or later process models use to interact with result processing model plug-ins. Because of the additional layers the process model plug-in architecture introduces, you can make changes to result processing functionality by modifying only the process model plug-in code or by creating a new custom process model plug-in without modifying the process model itself.



Although the TestStand 2012 or later process model plug-in architecture contains additional layers of abstraction between the process model and the actual result processing code, the underlying components that implement report generation and database logging are similar to the TestStand 2010 or earlier components. Even though some changes exist among the versions of the components, the TestStand 2012 or later components are compatible with the TestStand 2010 and 2012 process models.

Similarities between the TestStand 2012 or Later and TestStand 2010 Process Models

The TestStand 2012 or later process models maintain the following similarities with respect to the TestStand 2010 or earlier process model files:

- The same sequences and DLLs implement core functionality for report generation and database logging

- Report generation and database logging configuration options do not significantly change
- Callbacks defined in the process model file implement UUT & Serial Number tracking
- The `modelsupport2.dll` implements process model dialog boxes

Differences between the TestStand 2012 or Later and TestStand 2010 Process Models

Although many similarities exist, the TestStand 2012 or later process models include the following significant architecture differences:

- Process model plug-ins, not process model files, implement result processing
- Subsequences contain process model Execution Entry point functionality
- Modified existing components support the process model plug-in architecture

If you encounter an incompatibility in an existing test system when you use the TestStand 2012 or later process models or other circumstances prevent you from using the TestStand 2012 or later process models, you can use the equivalent legacy TestStand 2010 process models. You can use the Legacy Model Switcher tool to replace the TestStand 2012 or later process models with equivalent legacy TestStand 2010 process models.



Note

- If you migrate a custom legacy process model file to TestStand 2012, you must also migrate the corresponding `ModelSupport.seq` file from the earlier version of TestStand, or you must use the equivalent legacy TestStand 2010 process models, which include a legacy `ModelSupport.seq` file.
- This document reflects only the TestStand 2012 or later process models. If you are using the equivalent legacy TestStand 2010 process models, refer to the TestStand 2010 documentation for information about those process models.

Result Processing Improvements

You can use the built-in reporting, database, and offline results process model plug-ins to generate reports, log data to databases, and store results in compact offline result files for processing later, respectively. You can also create custom result processing plug-ins.

You can create multiple result processing configurations, which are named sets of instances of plug-ins you configure to accomplish a specific task, such as creating multiple reports with different formats or logging data to multiple databases. Creating multiple report processing configurations to use as the active configuration can help you quickly change how TestStand processes results. You can create result processing configurations to process results inline, meaning at run time, or offline. For example, you can specify to process offline results differently from inline results or you can specify to process offline result files differently when you store the files in different locations.

Select **Configure»Result Processing** to launch the default Result Processing dialog box, in which you enable or disable the installed built-in and custom result processing plug-ins in the active result processing configuration and also configure how the plug-ins process test results. You can insert multiple instances of the same plug-in with various configuration options.

TestStand Offline Results Processing Utility

You can use the TestStand Offline Results Processing Utility to process raw results files and generate reports at a time other than when sequence execution occurs or on a different computer, such as in situations in which performance is more important than generating immediate reports.

Related concepts:

- [Process Models](#)
- [Process Model Architecture](#)
- [Creating Process Model Plug-ins](#)
- [Dynamically Set the Client Sequence File](#)
- [Migrating Process Model Customizations to TestStand 2012 or Later](#)

Configuring Process Model Plug-ins

A process model invokes plug-in instances at defined points during execution. A plug-in instance specifies the plug-in sequence file to invoke and the set of options with which to invoke the sequence file. At run time, the process model reads the set of plug-in instances to invoke from the plug-in configuration (.cfg) files located in the `<TestStand Application Data>\Cfg\ModelPlugins` directory. By default, TestStand creates the `ResultProcessing.cfg` and `Addons.cfg` configuration files in the `<TestStand Application Data>\Cfg\ModelPlugins` directory if the files do not already exist.

ResultProcessing.cfg

The `ResultProcessing.cfg` file stores the information for plug-in instances you configure in the Result Processing dialog box.

Model Plug-in Add-ons and Addons.cfg

A model plug-in add-on is a model plug-in the process model invokes without requiring end users to create and configure a plug-in instance. For example, you can use an add-on to override the dialog boxes the Test UUTs and Single Pass Execution entry points launch and create a custom serial number entry solution instead.

Place a model plug-in sequence file in the `<TestStand Public>\Components\Models\ModelPlugins\Addons` directory to create a model plug-in add-on. The process model updates the `Addons.cfg` file when the model detects a new, removed, or updated plug-in in the `Addons` directory. The process model automatically uses the default set of options the add-on sequence file defines to create one plug-in instance for each add-on in the `Addons.cfg` file.

Creating Additional Model Plug-in Configuration Files

You can create additional model plug-in configuration files in the following ways:

- Call the `DisplayModelPluginDialogEx` function in `ModelSupport2.dll` as defined in `ResultProcessing.h`, located in the `<Teststand>\Components\Models\TestStandModels` directory, to

launch a dialog box that creates and edits a configuration file.

- Copy an existing configuration file to a new filename, open the new file in a text editor, and replace any occurrence of the previous filename with the new filename.
- Create a configuration file programmatically using an instance of the `PropertyObjectFile` class with a file type of `FileType_PropertyObjectFile`. The type of the `PropertyObjectFile.Data` property of the file must be `NI_ModelPluginConfigurationSet`.

Related concepts:

- [TestStand Directory Structure](#)
- [Programmatically Updating a Configuration File](#)
- [Model Plug-in Categories](#)

Programmatically Updating a Configuration File

If you deploy a new model plug-in to an existing TestStand installation and you want to automatically add an instance of the new plug-in to an existing configuration file, call the `AddResultProcessingPluginToConfiguration` or `AddModelPluginToConfiguration` functions in `ModelSupport2.dll` as defined in `ResultProcessing.h` located in the `<Teststand>\Components\Models\TestStandModels` directory.

You can call `RemoveResultProcessingPluginFromConfiguration` or `RemoveModelPluginFromConfiguration` to remove instances of a plug-in from a configuration file.

If you want to make arbitrary edits to an existing configuration file, you can load and save the file with the `PropertyObjectFile` interface. You can paste the following expression into a Statement step to create an example of how to modify a configuration file:

```
Locals.File =
RunState.Engine.NewPropertyObjectFile(FileType_PropertyObjectFile),
// don't forget to create Locals.File as an Object
Reference variable
```

```
Locals.File.AsPropertyObjectFile.Path =
```

```

RunState.Engine.GetTestStandPath(TestStandPath_Config) +
"\ModelPlugins\ResultProcessing.cfg",

Locals.File.AsPropertyObjectFile.ReadFile(ConflictHandler_Error),
// load the configuration file

// add expressions to change values as desired under:
Locals.File->Data.ModelPluginConfigurationSet

Locals.File.AsPropertyObjectFile.WriteFile(WriteFileFormat_Current)
// write back any changes to the configuration file

```

Related concepts:

- [TestStand Directory Structure](#)

Model Plug-in Categories

Each model plug-in file specifies a category name to determine when a plug-in can be invoked or made available. For example, in the Result Processing dialog box, you can insert only plug-in instances with the "ResultProcessor" category name. Similarly, the Offline Results Processing Utility invokes only plug-ins with the "ResultProcessor" category name. Therefore, do not use "ResultProcessor" as the category name for a plug-in you create that does not act as a result processor.

TestStand currently includes only results processing model plug-ins. You can use custom category names to define categories of model plug-ins you create that perform tasks other than results processing. For example, you might create categories of model plug-ins for automatically detecting and storing UUT serial numbers or for exchanging data with Enterprise Resource Planning systems.

Automatically Adding or Removing a Plug-in File to or from Configurations

If you plan to deploy a new model plug-in to an existing TestStand installation and you want TestStand to automatically add an instance of the new plug-in to existing configurations, you must set in the process model plug-in sequence file the Boolean `FileGlobals.ModelPluginComponentDescription.Default.Base.AutoAddToConfigurations` to True.

A setting of True causes the plug-in to be removed from all configurations when the plug-in is removed from the TestStand installation.

Alternatively, you can set the above Boolean to False to disable automatically updating the configurations.

Related concepts:

- [Configuring Process Model Plug-ins](#)
- [Programmatically Updating a Configuration File](#)
- [Model Plug-in Categories](#)

Creating Process Model Plug-ins

Process model plug-in sequence files must meet the following conditions:

- You must save the model plug-in sequence files in one of the following directories:
 - `<TestStand>\Components\Models\ModelPlugins` (built-in plug-ins)
 - `<TestStand Public>\Components\Models\ModelPlugins` (custom plug-ins)
 - `<TestStand>\Components\Models\ModelPlugins\Addons` (built-in add-ons)
 - `<TestStand Public>\Components\Models\ModelPlugins\Addons` (custom add-ons)
- You must include a file global variable named `ModelPluginComponentDescription` of type `NI_ModelPluginComponentDescription`
- You must select **Model** from the Type ring control on the Advanced tab of the Sequence File Properties dialog box for the model plug-in sequence file
- You must include one or more plug-in entry point sequences in the model plug-in sequence file

TestStand uses the location of the sequence file and the existence of the `ModelPluginComponentDescription` file global variable to identify plug-ins. Because TestStand loads and inspects all the sequence files within the top level of the `ModelPlugins` directories and of the `ModelPlugins\Addons` directories, you must place any supporting sequence files that a model plug-in calls in a different directory.

NI recommends that you use a unique prefix that includes your company name, such as `Acme_ReportGenerator.seq`, for custom plug-ins you create. TestStand uses the root filename as a <plug-in name> prefix for plug-in data types it creates, such as `Acme_ReportGeneratorOptions`. NI also recommends that you store any support files the plug-in requires in a subdirectory with the same name as the plug-in file, such as `ModelPlugins\Acme_ReportGenerator\Acme_ReportGenerator_Support.seq`.

Complete the following steps to generate a new plug-in that you can modify.

1. Select **Configure»Result Processing** to launch the Result Processing dialog box.
2. Enable the **Show More Options** option and click the **Advanced** button to launch the Advanced Result Processing Settings dialog box.
3. Click the **Create New Process Model Plug-in** button.
4. Use a unique filename that includes your company name as a prefix for the plug-in, such as `Acme_ReportGenerator.seq`.
5. Click **Save** to create and load the new plug-in sequence file. TestStand creates and adds the following new data types to the file. The data types are empty containers in which you can add plug-in-specific properties.
 - <plug-in name>Options
 - <plug-in name>AdditionalOptions
 - <plug-in name>RuntimeVariables
 - <plug-in name>PerSocketRuntimeVariables
6. Close the open dialog boxes to view and edit the plug-in you just created.
7. Use an expression in the `FileGlobals.ModelPluginComponentDescription.Default.Base.Display` subproperty to display a string that describes the plug-in or its output in the Output column of the Result Processing dialog box. Use an expression in the `FileGlobals.ModelPluginComponentDescription.DisplayNameExpression` subproperty to display a string that identifies the plug-in in the plug-in insertion menu of the Result Processing dialog box.
8. If you do not want plug-in end users to configure any plug-in-specific options, delete the Configure Standard Options and the Configure Additional Options sequences from the plug-in sequence file.
9. Complete the following steps if you want plug-in end users to configure plug-in-specific options.
 - Add each option you want plug-in end users to configure as a subproperty of

the <plug-in name>Options data type.

- Create a modal dialog box, in which plug-in end users can view and edit the plug-in options. Refer to the <TestStand Public>\Examples\Fundamentals\Launching a Modal Dialog directory for examples of creating modal dialog boxes.
 - Export a function, method, or VI to launch the dialog box and call the function, method, or VI in the Configure Standard Options sequence.
 - You can pass each <plug-in name>Options subproperty into and out of the dialog box entry point. You can also pass the plug-in or the <plug-in name>Options subproperty as a property object and use the TestStand PropertyObject class methods to access the <plug-in name>Options subproperties within the dialog box code.
 - Pass the value of the `RunState.Engine` property to the dialog box code so you can call the `Engine.NotifyStartOfModalDialogEx` and `Engine.NotifyEndOfModalDialog` methods to enforce the modality of the dialog box.
 - If you are using the LabWindows/CVI Adapter or the LabVIEW Adapter to create the dialog box, ensure that you configure the step that launches the dialog box to always run in the run-time engine to avoid creating out-of-process modal windows, which can result in hangs.



Note When you call a LabVIEW VI that creates ActiveX references, such as calling the `ApplicationMgr.GetEngine` method directly, you configure the VI to use the **same as caller** execution system, and you call the VI from a sequence running in a single-threaded apartment thread, LabVIEW might seem to hang with 100% processor use. Change the **Preferred Execution System** option in LabVIEW to a value other than **same as caller** to work around this issue.

- Optionally, you can use an expression in the `FileGlobals.ModelPluginComponentDescription.Default.Base.Opti` subproperty to display information in the Options column of the Result Processing dialog box. For example, the NI_ReportGenerator plug-in uses this expression to display the report format in the Options column of the Result Processing dialog box.

10. Consider whether you want the plug-in to always process information in a new

thread other than the model execution thread, never process information in a new thread, or allow end users to specify whether to process information in a new thread. Typically, you allow processing in a new thread if the plug-in requires a significant amount of processing time and the plug-in can process the information while processing information from other instances of the plug-in and testing subsequent UUTs. According to your decision, set the following subproperties of `FileGlobals.ModelPluginComponentDescription.Default`:

- `Base.NewThread`
- `Base.CompleteBeforeNextUUT`
- `Base.UseDefaultNewThreadImplementation`

11. Save and close the plug-in when you have completed the customizations.
12. Complete the following steps to add an instance of the new plug-in to a result processing configuration. TestStand stores the configuration in `<TestStand Application Data>\Cfg\ModelPlugins\ResultProcessing.cfg`. The process model loads this configuration file at run time to determine the plug-ins to invoke.
 - a. Select **Configure»Result Processing** to launch the Result Processing dialog box.
 - b. Click the **Add** button and select the plug-in name from the drop-down list.
13. When you have completed customizing and testing the plug-in, you can delete the entry point sequences the plug-in does not require. If you later determine that you need an entry point sequence you deleted, complete the following steps to add the entry point to the plug-in:
 - a. Unload the plug-in.
 - b. Restart the user interface or other TestStand-based application.
 - c. Generate a new plug-in with the same filename to a different location.
 - d. Copy the entry point sequences you need from the new plug-in to the original plug-in.
 - e. Delete the new plug-in. Alternatively, instead of deleting unused entry points, you can rename them so the model does not call them. For example, you can add a `[Reserved]` prefix the name of each unused entry point.

When you insert a plug-in instance, the modal dialog box inserts a copy of the `FileGlobals.ModelPluginComponentDescription.Default` property values to use for that instance of the plug-in. Any changes you make to these values in the plug-in sequence file apply only to subsequent instances of the plug-in you create.

Related concepts:

- [TestStand Directory Structure](#)
- [Model Plug-in Entry Points](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Programming with the TestStand API in LabWindows/CVI](#)
- [Dynamically Set the Client Sequence File](#)

Process Model Types

Model plug-ins typically do not need to determine the type of process model that invokes the plug-in. However, plug-in entry points can inspect the `ModelType` subproperty of the **ModelData** parameter to determine the type of process model that invokes the plug-in. Possible values of the `ModelType` field for the built-in process models include Sequential, Parallel, or Batch.

For example, if the value of the `ModelType` subproperty of the **ModelData** parameter is Batch, the built-in reporting model plug-in synchronizes its controller and socket threads to ensure it generates the batch report from the controller thread before generating UUT reports from test socket threads so that reports appear in the expected order when the threads write the reports to the same file.

Related concepts:

- [Model Plug-in Entry Points](#)

Process Model Thread Types

A process model can call plug-in entry points from multiple threads. The Batch process model and the Parallel process model create a separate thread and execution for each test socket and create a controller thread and execution to coordinate the activity of the socket threads. A process model calls some entry points from both socket and controller threads and other entry points only from socket threads or only from controller threads.

When a process model invokes a plug-in entry point, the model passes a parameter of type `NI_ModelThreadType` to the plug-in. The parameter includes the `IsController` and `IsTestSocket` Boolean subproperties. Plug-ins can use these

subproperties to determine the type of model thread that invokes the plug-in. For example, a `Begin` entry point might perform a general operation when the controller thread invokes the entry point or a socket-specific operation when a socket thread invokes the entry point.

The Sequential process model creates only one thread that coordinates and performs testing, so the Sequential process model always sets the `IsController` and `IsTestSocket` subproperties to `True`.

Plug-in instance properties that update at run time are called run-time variables. A plug-in uses the following subproperties to store run-time variables specific to the plug-in:

- `<NI_ModelPlugin>.PluginSpecific.RuntimeVariables` of type `<plug-in name>RuntimeVariables`
- `<NI_ModelPlugin>.PluginSpecific.RuntimeVariables.PerSocket` of type array of `<plug-in name>PerSocketRuntimeVariables`

The process model sets the `PerSocket` property array to have one element for each test socket. Therefore, you can add a property to the `<plug-in name>PerSocketRuntimeVariables` data type to create a run-time variable that has a separate value for each socket thread.

When you use the Batch or Parallel process models to invoke a plug-in, the plug-in executes as a multithreaded sequence because the model invokes plug-in entry points in the controller thread and the socket threads. Therefore, you can access properties of the plug-in instance from multiple threads in parallel. TestStand ensures that read and write operations to TestStand property values and array element values are atomic. However, NI recommends that you use the following strategies to ensure that different threads safely share access to plug-in instance variables:

- Add run-time variables only to the `<plug-in name>RuntimeVariables` and `<plug-in name>PerSocketRuntimeVariables` data types.
- If a run-time variable must have a unique value for each test socket, add the variable to the `<plug-in name>PerSocketRuntimeVariables` data type.
- Use a TestStand lock or other synchronization method to ensure a thread has exclusive access to the plug-in run-time variables when required.

Related concepts:

- [Model Plug-in Entry Points](#)
- [Batch Controller and Test Socket Synchronization Architecture](#)
- [Avoid Batch Synchronization Sections in Process Model Plug-in Entry Points](#)
- [Thread Safety of the PropertyObject API and TestStand Variables](#)

Step Results

A model plug-in can receive step results after a UUT completes or on-the-fly.

After UUT Completion

After each UUT completes, the process model calls the Model Plugin – UUT Done entry point and uses the **MainSequenceResult** parameter to pass a reference to the result of the process model Sequence Call step that calls the client file. The `Parameters.MainSequenceResult.TS.SequenceCall.ResultList` array contains the step results of the top-level client sequence. If a result in the array is a Sequence Call step result, the `TS.SequenceCall.ResultsList` property of the subsequence includes the results of all steps in the subsequence. The `TS.SequenceCall.Sequence`, `TS.SequenceCall.SequenceFile`, and `TS.SequenceCall.Status` properties include the sequence name, sequence file path, and result status for the subsequence.

A plug-in that processes results, such as a report generator or a database logger, typically recursively traverses the `TS.SequenceCall.ResultsList` array, including results of subsequences, to generate a report or log results to a database.

On-The-Fly

If you enable a plug-in to process results on-the-fly by setting the `ProcessOnTheFly` subproperty to `True`, TestStand progressively collects results concurrent with the test execution. When TestStand collects a result and exceeds a threshold in time or in the number of collected results, the process model passes the results collected since the last time TestStand exceeded the threshold to the Model Plugin – OnTheFly Step Results entry point for processing.

A plug-in that processes results on-the-fly, such as a report generator or a database

logger, typically appends data to a report or logs the data to a database while iterating through each array of results the model passes to the OnTheFly Step Results entry point.

On-The-Fly - Provisional Results

TestStand cannot determine the result status of a Sequence Call step before the subsequence completes. Therefore, TestStand does not generate the final parent result for the steps of a subsequence until the subsequence completes. To support process model plug-ins that must immediately identify the parent result for each child result, TestStand generates a provisional result for a Sequence Call step before executing the steps in the subsequence.

Similarly, TestStand cannot determine the result status of a step that loops before all the loop iterations in the step complete. Therefore, TestStand does not generate the final step result of a step that loops until all loop iterations complete. To support process model plug-ins that display loop information, such as the loop index, the number of loops that passed, and the number of loops that failed, while the step loops and before loop iterations complete, TestStand generates a provisional result for steps that loop.

The ID of a provisional result is the same as the ID of the corresponding final result. TestStand passes provisional results only to the Model Plugin - OnTheFly Step Results plug-in entry point and PostResults engine callback. Use the existence of the `TS.Provisional` Boolean subproperty to identify provisional results.

Additionally, TestStand generates a provisional result after executing all the steps in a sequence that executes in a new thread. Some process model plug-ins can use this provisional result to re-process results collected for steps that executed in a new thread. The ID of the provisional result is the same as the ID of the corresponding Sequence Call step that executed the sequence. Use the existence of the `TS.ThreadComplete` Boolean subproperty to identify this provisional result.

In a Model Plugin - OnTheFly Step Results plug-in entry point, you use or ignore provisional results depending on what the plug-in requires.

Related concepts:

- [Model Plugin – UUT Done](#)
- [Model Plugin – OnTheFly Step Results](#)
- [Executions](#)

Structure of Plug-in Sequence Files

Model plug-ins include specific entry points, entry point parameters, and data types.

Related concepts:

- [Model Plug-in Entry Points](#)
- [Plug-in Entry Point Parameters](#)

Model Plug-in Entry Points

TestStand process models invoke model plug-ins by calling the following set of optional entry point sequences with a pre-defined set of parameters at specific points in the testing process.

- Model Plugin – Configure Standard Options
- Model Plugin – Configure Additional Options
- Model Plugin – Initialize
- Model Plugin – Begin
- Model Plugin – Pre Batch
- Model Plugin – Pre UUT
- Model Plugin – Batch Start
- Model Plugin – UUT Start
- Model Plugin – OnTheFly Step Results
- Model Plugin – UUT Done
- Model Plugin – Batch Done
- Model Plugin – Post UUT
- Model Plugin – Post Batch
- Model Plugin – End

To simplify the model plug-ins you create, implement only the entry points you need to perform specific tasks. The Configure Standard Options and the Configure Additional Options entry points provide functionality you invoke to allow end users to edit a model plug-in configuration. Process models invoke the remaining entry points

at run time from the locations shown in bold text in the following tables.



Note The run-time order is the same for the Test UUTs and Single Pass entry points except that the Single Pass entry point does not loop on UUTs or batches.

Sequential Model Plug-in Entry Point Run-Time Order for Controller and Socket Threads

Model Plugin - Initialize

ModelOptions Callback

Model Plugin - Begin

Process Setup

PreUUTLoop Callback

[Begin UUT Loop]

PreUUT Callback

Model Plugin - Pre UUT

If (!continueTesting) goto [End of UUT Loop]

Model Plugin - UUT Start

PreMainSequence Callback

MainSequence Callback

PostMainSequence Callback

Model Plugin - OnTheFly Step Results

Model Plugin - UUT Done

Model Plugin - Post UUT

PostUUT Callback

[End of UUT Loop]

Sequential Model Plug-in Entry Point Run-Time Order for Controller and Socket Threads

PostUUTLoop Callback

Process Cleanup Callback

Model Plugin - End



Note In the following tables, all operations in a single row occur before all operations in the next row, and all operations within a cell occur in the order indicated. Operations in the same row but in different columns can occur in any order.

Parallel Model Plug-in Entry Point Run-Time Order	
Controller Thread	Test Socket Threads
Model Plugin - Initialize ModelOptions Callback Model Plugin - Begin Process Setup	
	Model Plugin - Begin PreUUTLoop Callback [Begin UUT Loop] PreUUT Callback Model Plugin - Pre UUT If (!continueTesting) goto [End of UUT Loop] Model Plugin - UUT Start

Parallel Model Plug-in Entry Point Run-Time Order	
Controller Thread	Test Socket Threads
	PreMainSequence Callback MainSequence Callback PostMainSequence Callback Model Plugin - OnTheFly Step Results Model Plugin - UUT Done Model Plugin - Post UUT PostUUT Callback [End of UUT Loop] PostUUTLoop Callback Model Plugin - End
Process Cleanup Callback Model Plugin - End	

Batch Model Plug-in Entry Point Run-Time Order	
Controller Thread	Test Socket Threads
Model Plugin - Initialize ModelOptions Callback Model Plugin - Begin Process Setup	

Batch Model Plug-in Entry Point Run-Time Order	
Controller Thread	Test Socket Threads
PreBatchLoop Callback	
	Model Plugin - Begin PreUUTLoop Callback
[Begin Batch Loop] PreBatch Callback Model Plugin - Pre Batch If (!continueTesting) goto [End of Batch Loop]	
Model Plugin - Batch Start	[Begin UUT Loop] PreUUT Callback Model Plugin - Pre UUT If (!continueTesting) goto [End of UUT Loop] Model Plugin - UUT Start
	PreMainSequence Callback MainSequence Callback PostMainSequence Callback Model Plugin - OnTheFly Step Results
Model Plugin -Batch Done	Model Plugin - UUT Done

Batch Model Plug-in Entry Point Run-Time Order	
Controller Thread	Test Socket Threads
	Model Plugin - Post UUT PostUUT Callback
Model Plugin - Post Batch PostBatch Callback	
[End of Batch Loop]	[End of UUT Loop] PostUUTLoop Callback Model Plugin - End
PostBatchLoop Callback Process Cleanup Callback Model Plugin - End	

Related concepts:

- [Plug-in Entry Point Parameters](#)
- [Model Plugin – Configure Standard Options](#)
- [Model Plugin – Configure Additional Options](#)
- [Model Plugin – Initialize](#)
- [Model Plugin – Begin](#)
- [Model Plugin – Pre Batch](#)
- [Model Plugin – Pre UUT](#)
- [Model Plugin – Batch Start](#)
- [Model Plugin – UUT Start](#)
- [Model Plugin – OnTheFly Step Results](#)

- [Model Plugin – UUT Done](#)
- [Model Plugin – Batch Done](#)
- [Model Plugin – Post UUT](#)
- [Model Plugin – Post Batch](#)
- [Model Plugin – End](#)
- [Batch Controller and Test Socket Synchronization Architecture](#)

Model Plugin – Configure Standard Options

Use the Configure Standard Options entry point to provide a way to configure option settings for an instance of a plug-in. A plug-in that implements Configure Standard Options includes an icon in the Options column of the Result Processing dialog box. Model plug-in end users click the icon to invoke Configure Standard Options, which launches a modal dialog box in which the end user can view and edit options for the instance of the plug-in.

Use the `Parameters.ModelPlugin.PluginSpecific.Options` subproperty to store the plug-in-specific options for end users to configure in the dialog box the entry point launches.



Note If the plug-in implements the OnTheFly Step Results entry point and you want to provide a way to specify whether to process results on-the-fly, include a Boolean on-the-fly option in the dialog box and store the value of the option in the `ModelPlugin.Base.ProcessOnTheFly` Boolean property, not a plug-in-specific property.

The Configure Standard Options entry point accepts the following parameters:

Parameter Name	Data Type
Changed	Boolean
ProcessModel	Reference
ModelType	NI_ModelType
ModelPlugin	NI_ModelPlugin

Related concepts:

- [Model Plugin – OnTheFly Step Results](#)
- [Plug-in Entry Point Parameters](#)

Model Plugin – Configure Additional Options

Use the Configure Additional Options entry point to launch a second dialog box to provide a way for plug-in end users to configure option settings for the instance of a plug-in. Typically, you do not need this entry point because you can present options in a single dialog box by using tabs, modal popup dialog boxes, or other organizational methods to accommodate many options.

Use this entry point when you create a new plug-in by extending an existing plug-in and you want to provide a small number of additional options for the new plug-in without changing the options dialog box for the original plug-in. Configure Additional Options launches a second dialog box that contains options specific to the new plug-in, and the Configure Standard Options entry point launches the unchanged options dialog box from the original plug-in.

If a plug-in implements Configure Additional Options, the Options column of the Result Processing dialog box includes a second settings icon immediately to the right of the icon that invokes Configure Standard Options.

Use the

`Parameters.ModelPlugin.PluginSpecific.AdditionalOptions` subproperty to store the additional options for the enhanced plug-in.

The Configure Additional Options entry point accepts the following parameters, which are the same as the Configure Standard Options entry point:

Parameter Name	Data Type
Changed	Boolean
ProcessModel	Reference
ModelType	NI_ModelType

Parameter Name	Data Type
ModelPlugin	NI_ModelPlugin

Related concepts:

- [Model Plugin – Configure Standard Options](#)
- [Plug-in Entry Point Parameters](#)

Model Plugin – Initialize

A process model calls the Initialize entry point once in its controller thread before calling any other run-time entry points and before calling the ModelPluginOptions and ModelPluginConfiguration callbacks.

Typically, you implement Initialize only if you want to initialize variables or options before the process model calls the ModelPluginOptions and ModelPluginConfiguration callbacks. Because these callbacks can disable or remove plug-ins, no guarantee exists that the process model calls the End entry point for every plug-in that calls Initialize. Therefore, use the Begin entry point to perform any initialization that requires cleanup in End.

Do not use Initialize to perform initialization that depends on the values of model plug-in options because the subsequent ModelPluginOptions and ModelPluginConfiguration callbacks can alter the option values. Instead, use the Begin entry point to perform option-dependent initialization.

Initialize is the only run-time entry point in which you can change the enabled state of a plug-in instance.

Set the value of the

`FileGlobals.ModelPluginComponentDescription.Default.Base.AlwaysInit` property to `True` for the process model to call the Initialize entry point and the ModelPluginOptions callback even if you do not enable the instance of the plug-in so that the plug-in and the client sequence file can dynamically enable or disable the instances of the plug-in.

If you set the value of the `AlwaysInitialize` property to `False` and the `ModelPluginConfiguration` callback enables a disabled instance of the plug-in, the process model calls the `Initialize` entry point and the `ModelPluginOptions` callback after the `ModelPluginConfiguration` callback returns.

The `Initialize` entry point accepts the following parameters:

Parameter Name	Data Type
<code>ModelPlugin</code>	<code>NI_ModelPlugin</code>
<code>ModelPluginConfiguration</code>	<code>NI_ModelPluginConfiguration</code>
<code>ModelData</code>	<code>Container</code>

Related concepts:

- [Process Model Thread Types](#)
- [Model Plug in Client-File Callbacks](#)
- [Model Plugin – End](#)
- [Model Plugin – Begin](#)
- [Plug-in Entry Point Parameters](#)

Model Plugin – Begin

Use the `Begin` entry point to perform required setup or initialization tasks, such as opening files or establishing database connections.

A process model calls `Begin` once in its controller thread and once in each test socket thread after calling the `ModelPluginOptions` and `ModelPluginConfiguration` callbacks. In contrast, a process model calls the `Initialize` entry point only in the controller thread and before calling the `ModelPluginOptions` and `ModelPluginConfiguration` callbacks.

Process models complete the call to the `Begin` from the controller thread before calling `Begin` from any socket thread.

The `Begin` entry point accepts the following parameters:

Parameter Name	Data Type
ModelPlugin	NI_ModelPlugin
ModelPluginConfiguration	NI_ModelPluginConfiguration
ModelThreadType	NI_ModelThreadType
ModelData	Container
StartDate	DateDetails
StartTime	TimeDetails
ProcessModelClientPath	Path
ParentThread	Reference

Related concepts:

- [Process Model Thread Types](#)
- [Model Plug in Client-File Callbacks](#)
- [Model Plugin – Initialize](#)
- [Plug-in Entry Point Parameters](#)

Model Plugin – Pre Batch

Use the Pre Batch entry point to identify the batch or determine whether to continue testing.

The Batch process model calls Pre Batch in the controller thread before each batch. The Batch process model completes all calls to the Begin entry point before calling Pre Batch. The Batch process model completes the call to Pre Batch from the controller thread before calling the Pre UUT entry point from any socket thread. The Parallel and Sequential process models do not call Pre Batch.

The Pre Batch entry point accepts the following parameters, which are the same as the Pre UUT entry point:

Parameter Name	Data Type
ModelPlugin	NI_ModelPlugin

Parameter Name	Data Type
ModelPluginConfiguration	NI_ModelPluginConfiguration
ModelThreadType	NI_ModelThreadType
ModelData	Container
ProcessModelClientPath	Path
StartDate	DateDetails
StartTime	TimeDetails
UUT	UUT
ContinueTesting	Boolean

Related concepts:

- [Process Model Thread Types](#)
- [Model Plugin – Begin](#)
- [Model Plugin – Pre UUT](#)
- [Plug-in Entry Point Parameters](#)

Model Plugin – Pre UUT

Use the Pre UUT entry point to identify the UUT or determine whether to continue testing.

A process model calls Pre UUT in the test socket thread before testing each UUT.

The Batch process model completes the call to the Pre Batch entry point before calling Pre UUT for any test socket.

The Pre UUT entry point accepts the following parameters:

Parameter Name	Data Type
ModelPlugin	NI_ModelPlugin
ModelPluginConfiguration	NI_ModelPluginConfiguration

Parameter Name	Data Type
ModelThreadType	NI_ModelThreadType
ModelData	Container
ProcessModelClientPath	Path
StartDate	DateDetails
StartTime	TimeDetails
UUT	UUT
ContinueTesting	Boolean

If the **ContinueTesting** parameter is `False` after the process model calls Pre UUT for every plug-in, the process model exits the UUT loop without calling the UUT Start, UUT Done, or Post UUT entry points. If multiple model plug-ins use conflicting values for **ContinueTesting**, TestStand uses the value set by the last plug-in to execute.

Related concepts:

- [Process Model Thread Types](#)
- [Model Plugin – Pre Batch](#)
- [Plug-in Entry Point Parameters](#)
- [Model Plugin – UUT Start](#)
- [Model Plugin – UUT Done](#)
- [Model Plugin – Post UUT](#)

Model Plugin – Batch Start

Use the Batch Start entry point to perform setup or initialization tasks before testing each batch.

The Batch process model calls Batch Start in the controller thread after calling the Pre Batch entry point. The Batch process model runs Batch Start and the UUT Start entry point in parallel. The Batch process model completes the call to Batch Start before calling the UUT Done entry point in any socket. The Parallel and Sequential process models do not call Batch Start.

The Batch Start entry point accepts the following parameters, which are the same as the UUT Start entry point:

Parameter Name	Data Type
ModelPlugin	NI_ModelPlugin
ModelPluginConfiguration	NI_ModelPluginConfiguration
ModelThreadType	NI_ModelThreadType
ModelData	Container
ProcessModelClientPath	Path
StartDate	DateDetails
StartTime	TimeDetails
UUT	UUT

Related concepts:

- [Process Model Thread Types](#)
- [Model Plugin – Pre Batch](#)
- [Model Plugin – UUT Start](#)
- [Model Plugin – UUT Done](#)
- [Plug-in Entry Point Parameters](#)

Model Plugin – UUT Start

Use the UUT Start entry point to perform setup or initialization tasks before testing each UUT. For example, a plug-in might use UUT Start to open per-UUT report files.

A process model calls UUT Start in the test socket thread after calling the Pre UUT entry point but before testing the UUT.

The Batch process model runs UUT Start and the Batch Start entry point in parallel.

The UUT Start entry point accepts the following parameters, which are the same as Pre UUT except for the **ContinueTesting** parameter:

Parameter Name	Data Type
ModelPlugin	NI_ModelPlugin
ModelPluginConfiguration	NI_ModelPluginConfiguration
ModelThreadType	NI_ModelThreadType
ModelData	Container
ProcessModelClientPath	Path
StartDate	DateDetails
StartTime	TimeDetails
UUT	UUT

Related concepts:

- [Process Model Thread Types](#)
- [Model Plugin – Pre UUT](#)
- [Model Plugin – Batch Start](#)
- [Plug-in Entry Point Parameters](#)

Model Plugin – OnTheFly Step Results

Use the OnTheFly Step Results entry point to obtain step results during test execution. For example, when you enable the On-The-Fly Reporting option in the Report Options dialog box, the built-in reporting plug-in calls OnTheFly Step Results to append results information to the report during test execution.

A process model calls OnTheFly Step Results only for plug-ins configured to process results on-the-fly. The `<NI_ModelPlugin>.Base.ProcessOnTheFly` subproperty stores the on-the-fly setting. TestStand collects step results until the number of results exceeds a threshold value you specify and then passes the results through the process model to OnTheFly Step Results for processing. The number of results the entry point receives can vary depending on multiple factors, such as the options you specify in the Advanced Result Processing Settings dialog box.

A process model calls OnTheFly Step Results from all threads that generate results.

Controller threads in the Batch and Parallel process models do not generate results.

Although a test socket thread can create multiple child threads, the TestStand process models prevent sockets from calling OnTheFly Step Results simultaneously for results from more than one thread in the same test socket. Therefore, OnTheFly Step Results can access per-socket data without additional synchronization.

The OnTheFly Step Results entry point accepts the following parameters:

Parameter Name	Data Type
ModelPlugin	NI_ModelPlugin
ModelData	Container
StartTime	TimeDetails
StartDate	DateDetails
UUT	UUT
Context	Reference
Steps	Reference Array
Results	Reference Array
CallbackNames	String Array
ParentIds	Numeric Array
ModelPluginConfiguration	NI_ModelPluginConfiguration

The **Steps**, **Results**, **CallbackNames**, and **ParentIds** parameters are parallel arrays. For any given array index n , the result element at `Results[n]` and the values of `Steps[n]`, `CallbackNames[n]`, and `ParentIds[n]` represent the n^{th} result and the step that generated the result, the callback name of the result, and the ID of the parent for that result, respectively.

Related concepts:

- [Process Model Thread Types](#)
- [Plug-in Entry Point Parameters](#)
- [Subsequence Results](#)

Model Plugin – UUT Done

Use the UUT Done entry point to perform an action after each UUT completes. For example, a plug-in might process the UUT test results and store the information in a report file or database tables.

A process model calls UUT Done in each test socket thread after testing the UUT.

The Batch process model runs UUT Done and the Batch Done entry point in parallel.

Set the `<NI_ModelPlugin>.Base.NewThread` and `<NI_ModelPlugin>.Base.UseDefaultNewThreadImplementation` properties to `True` for the process model to invoke UUT Done in a new thread. When the process model invokes UUT Done in a new thread, it passes all parameters by value except for **ModelPlugin** and **ModelPluginConfiguration**.

The UUT Done entry point accepts the following parameters, which are the same as the UUT Start entry point and include the additional **UUTStatus**, **MainSequenceResult**, **CallingThread**, **CallingThreadBlocker**, and **PriorIterationNewThread** parameters:

Parameter Name	Data Type
CallingThread	Reference
CallingThreadBlocker	Reference
MainSequenceResult	Reference
ModelPlugin	NI_ModelPlugin
ModelPluginConfiguration	NI_ModelPluginConfiguration
ModelThreadType	NI_ModelThreadType
ModelData	Container
PriorIterationNewThread	Reference
ProcessModelClientPath	Path
StartDate	DateDetails
StartTime	TimeDetails

Parameter Name	Data Type
UUT	UUT
UUTStatus	String

Related concepts:

- [Process Model Thread Types](#)
- [Model Plugin – Batch Done](#)
- [Plug-in Entry Point Parameters](#)
- [Model Plugin – UUT Start](#)

Model Plugin – Batch Done

Use the Batch Done entry point to perform an action after each batch completes. For example, a plug-in might generate a batch report that summarizes the results of all test socket threads.

The Batch process model runs Batch Done and the UUT Done entry point in parallel. The Batch process model completes all calls to the UUT Start entry point before calling Batch Done. The Sequential and Parallel process models do not call Batch Done.

Set the `<NI_ModelPlugin>.Base.NewThread` and `<NI_ModelPlugin>.Base.UseDefaultNewThreadImplementation` properties to `True` for the process model to invoke Batch Done in a new thread. When the process model invokes Batch Done in a new thread, it passes all parameters by value except for **ModelPlugin** and **ModelPluginConfiguration**.

The Batch Done entry point accepts the following parameters, which are the same as the UUT Done entry point:

Parameter Name	Data Type
CallingThread	Reference
CallingThreadBlocker	Reference
MainSequenceResult	Reference

Parameter Name	Data Type
ModelPlugin	NI_ModelPlugin
ModelPluginConfiguration	NI_ModelPluginConfiguration
ModelThreadType	NI_ModelThreadType
ModelData	Container
PriorIterationNewThread	Reference
ProcessModelClientPath	Path
StartDate	DateDetails
StartTime	TimeDetails
UUT	UUT
UUTStatus	String

Related concepts:

- [Model Plugin – UUT Done](#)
- [Model Plugin – UUT Start](#)
- [Plug-in Entry Point Parameters](#)

Model Plugin – Post UUT

A process model calls the Post UUT entry point after calling the UUT Done entry point.

Post UUT is similar to UUT Done except that Post UUT entry point never runs in a new thread and the Batch process model does not call the Post Batch entry point until every call to Post UUT completes.

Post UUT accepts the following parameters, which are the same as UUT Done except for the **CallingThread**, **CallingThreadBlocker**, and **PriorIterationNewThread** parameters:

Parameter Name	Data Type
MainSequenceResult	Reference

Parameter Name	Data Type
ModelPlugin	NI_ModelPlugin
ModelPluginConfiguration	NI_ModelPluginConfiguration
ModelThreadType	NI_ModelThreadType
ModelData	Container
ProcessModelClientPath	Path
StartDate	DateDetails
StartTime	TimeDetails
UUT	UUT
UUTStatus	String

Related concepts:

- [Model Plugin – UUT Done](#)
- [Model Plugin – Post Batch](#)
- [Plug-in Entry Point Parameters](#)

Model Plugin – Post Batch

A process model calls the Post Batch entry point after calling the Batch Done entry point.

Post Batch is similar to Batch Done except that Post Batch never runs in a new thread and the Batch process model does not call Post Batch until every call to the Post UUT entry point completes.

The Sequential and Parallel process models do not call Post Batch.

The Post Batch entry point accepts the following parameters, which are the same as the Post UUT entry point:

Parameter Name	Data Type
MainSequenceResult	Reference

Parameter Name	Data Type
ModelPlugin	NI_ModelPlugin
ModelPluginConfiguration	NI_ModelPluginConfiguration
ModelThreadType	NI_ModelThreadType
ModelData	Container
ProcessModelClientPath	Path
StartDate	DateDetails
StartTime	TimeDetails
UUT	UUT
UUTStatus	String

Related concepts:

- [Model Plugin – Batch Done](#)
- [Model Plugin – Post UUT](#)
- [Plug-in Entry Point Parameters](#)

Model Plugin – End

Use the End entry point to perform required cleanup or finalization tasks, such as closing files or database connections.

A process model calls End once in its controller thread and once in each test socket thread. Process models call End from the controller thread only after all calls to End from test socket threads complete. End is the last entry point the process model calls at run time after the UUT loop, if any, exits.

The End entry point accepts the following parameters:

Parameter Name	Data Type
ModelPlugin	NI_ModelPlugin
ModelPluginConfiguration	NI_ModelPluginConfiguration

Parameter Name	Data Type
ModelThreadType	NI_ModelThreadType
ModelData	Container

Related concepts:

- [Process Model Thread Types](#)
- [Plug-in Entry Point Parameters](#)

Order of Entry Point Execution at Run Time

The process model calls each plug-in entry point in the order in which the plug-in instance appears in the corresponding configuration file. For result processors, the order in which the plug-in instances appear in the Result Processing dialog box determines the order in which the instances appear in the corresponding configuration file.

When multiple configuration files define plug-in instances, the values of the `<NI_ModelPluginConfiguration>.RunOrder` property in each configuration file define the order in which the process model invokes the group of plug-in instances from each file. By default, the `Addons.cfg` configuration file contains a `<NI_ModelPluginConfiguration>.RunOrder` value of `-10.0`, which causes add-on plug-ins to run before result processor plug-ins in the `ResultProcessing.cfg` file, which uses the default value of `0.0` for the `<NI_ModelPluginConfiguration>.RunOrder` property.

Additionally, if the values of the `<NI_ModelPlugin>.Base.RunOrder` properties of plug-in instances found in each configuration file are not equal, the process models invoke plug-in instances in ascending order based on the value of the `<NI_ModelPlugin>.Base.RunOrder` property of the plug-in instance found in the corresponding configuration file.



Note Plug-in configuration files store a snapshot of the plug-in instance. For result processing plug-ins, the Configure Standard Options or Configure Additional Options entry points for your plug-in must provide an interface for

users to modify the `<NI_ModelPlugin>.Base.RunOrder` property, which is stored in the `Base.RunOrder` parameter of the `Configure Standard Options` and `Configure Additional Options` entry points. You must modify the `ModelPluginComponentDescription.Default.Base.RunOrder` file global variable for add-ons.

TestStand always calls the UUT Done and Batch Done entry points of plug-in instances configured to use a new thread before calling UUT Done and Batch Done entry points of plug-in instances not configured to use a new thread. TestStand uses the same guidelines to determine the calling order for UUT Done and Batch Done entry points of plug-in instances configured to use a new thread as well as the calling order for UUT Done and Batch Done entry points of plug-in instances not configured to use a new thread.

Plug-in Entry Point Parameters

Model plug-in entry points accept the following parameters:

Name	Data Type	Model Plug-in Entry Point	Description
CallbackNames	String Array	OnTheFly Step Results	Each element in this array specifies a <code>Parameters.Results</code> array. <code>SequenceCall</code> for steps that <code>TestStand</code> calls. <code>SequenceCall</code> steps.
CallingThread	Reference	UUT Done Batch Done	A reference to the test socket or command line parameter that specifies the model plug-in to run in a new thread. The parameter specifies.
CallingThreadBlocker	Reference	UUT Done Batch	Reference to a TestStand notification object that is created when executing the entry point than is used to prevent the process from proceeding until the entry point sets the <code>IsBlocked</code> property to <code>True</code> . The <code>IsBlocked</code> property is subject to change if the process runs.

Name	Data Type	Model Plug-in Entry Point	Description
		Done	completes. For example, the UUT <code>ResultProcessor.Runtime</code> UUT Done entry point in the built notification.
Context	SequenceContext	OnTheFly Step Results	A reference to the <code>SequenceContext</code> Results entry point. Because <code>TestStand</code> OnTheFly Step Results entry point alternative implementation is possible <code>FileGlobals.ResultProcessor</code> property of the plug-in to <code>False</code> results OnTheFly Step Results results sequences. If you must access the <code>RunState.Execution.Post</code> <code>RunState.Execution.Post</code> the process model invokes OnTheFly you change settings that determine such that the process model calls another plug-in instance sets and
ContinueTesting	Boolean (output)	Pre Batch Pre UUT	If this parameter is <code>False</code> after the UUT loop without calling the UUT model calls the Pre Batch entry point Start, Batch Done, or Post Batch UUT Start, UUT Done, or Post UUT TestStand uses the value set by the
MainSequenceResult	Reference	Post UUT UUT Done Post Batch Batch Done	A reference to the result of the process client sequence file <code>MainSequence</code> property. TestStand passes <code>Nothing</code>

Name	Data Type	Model Plug-in Entry Point	Description
ModelData	Container  Note The actual data type varies at run time depending on the process model. For built-in process models, the data type can be NI_SequentialModelData, NI_ParallelModelData2, or NI_BatchModelData2.	Begin Pre Batch Pre UUT Batch Start UUT Start OnTheFly Step Results UUT Done Batch Done Post UUT Post Batch End	Contains model-specific information
ModelPlugin	NI_ModelPlugin	<All>	Configuration and run-time variables
ModelPluginConfiguration	NI_ModelPluginConfiguration	Begin Pre Batch Pre UUT Batch Start	Configuration and run-time variables within the ModelPluginConfiguration

Name	Data Type	Model Plug-in Entry Point	Description
		UUT Start UUT Done Batch Done Post UUT Post Batch End	
ModelThreadType	NI_ModelThreadType	Begin Pre Batch Pre UUT Batch Start UUT Start UUT Done Batch Done Post UUT Post Batch End	Specifies whether the current thr

Name	Data Type	Model Plug-in Entry Point	Description
ModelType	NI_ModelType	Configure Standard Options Configure Additional Options	String that indicates the type of test model. The model can use this value to determine its behavior.
ParentIds	Numeric Array	OnTheFly Step Results	Each element in this array specifies the parent ID of the <code>Parameters.Results</code> array. The parent ID is the result. You can use parent IDs to identify the parent result of a model <code>MainSequence</code> callback. If a result does not have a parent result. In this case, the parent ID is 0. <code>TestStand</code> negates the value so you can identify the parent result.
ParentThread	Reference	Begin	When a test socket thread calls the <code>Begin</code> method, the <code>ParentThread</code> parameter. If the calling thread is the test socket thread, the <code>ParentThread</code> is also the test socket thread.
PriorIterationNewThread	Reference	UUT Done Batch Done	A reference to the thread that ran the previous iteration. You can invoke the previous call in a new thread. For example, if you configure a test to run in parallel, it waits for this thread to complete before running in chronological order.
ProcessModelClientPath	Path	Begin Pre Batch Pre UUT Batch Start UUT Start UUT Done	Absolute path to the client sequence file.

Name	Data Type	Model Plug-in Entry Point	Description
		Batch Done Post UUT Post Batch	
Results	Array of References	OnTheFly Step Results	An array of references to the step OnTheFly Step Results entry point and its reference.
StartDate	DateDetails	Begin Pre Batch Pre UUT Batch Start UUT Start OnTheFly Step Results UUT Done Batch Done Post UUT Post	When passed to the Begin entry point, the StartDate entry point. Otherwise, the StartDate current batch began testing for a

Name	Data Type	Model Plug-in Entry Point	Description
		Batch	
StartTime	TimeDetails	Begin Pre Batch Pre UUT Batch Start UUT Start OnTheFly Step Results UUT Done Batch Done Post UUT Post Batch	When passed to the Begin entry point, the plug-in serializes the test results. Otherwise, the current batch began testing for
Step	Array of References	OnTheFly Step Results	An array of references to the step element in the array is a TestStart. To serialize the steps to offline results, the plug-in uses the Step parameter, <code>FileGlobals.ResultProcessing</code> property of the plug-in to <code>False</code> .

Name	Data Type	Model Plug-in Entry Point	Description
UUT	UUT	Pre Batch Pre UUT Batch Start UUT Start OnTheFly Step Results UUT Done Batch Done Post UUT Post Batch	For the Pre UUT, UUT Start, UUT Done, Batch Done, and Post UUT entry points, the UUT object contains UUT information for the current test session.
UUTStatus	String	UUT Done Post UUT Batch Done Post Batch	The result status of the UUT, such as UUT Done, Batch Done, and Post Batch entry points, and the UUTStatus object contains the result status of the UUT.

Related concepts:

- [Model Plugin – OnTheFly Step Results](#)

- [Subsequence Results](#)
- [Model Plugin – UUT Done](#)
- [Model Plugin – Batch Done](#)
- [Process Model Thread Types](#)
- [Model Plugin – UUT Start](#)
- [Model Plugin – Begin](#)
- [Model Plugin – Pre Batch](#)
- [Model Plugin – Pre UUT](#)
- [Model Plugin – Post UUT](#)
- [Model Plugin – Batch Start](#)
- [Model Plugin – Post Batch](#)
- [Model Plugin – End](#)
- [Model Plugin – Configure Standard Options](#)
- [Model Plugin – Configure Additional Options](#)
- [Thread Safety of the PropertyObject API and TestStand Variables](#)

Avoid Batch Synchronization Sections in Process Model Plug-in Entry Points

Do not use Serial, Parallel, or One Thread Only batch synchronization sections in model plug-in entry point sequence files.

Related concepts:

- [Model Plug-in Entry Points](#)

Model Plug in Client-File Callbacks

A client file can override the ModelPluginOptions or the ModelPluginConfiguration model callback to inspect or alter the plug-in options an end user configures.

ModelPluginOptions

The model calls the ModelPluginOptions callback once for each plug-in instance and passes the instance to the **ModelPlugin** parameter. Use the subproperties of the `Parameters.ModelPlugin.PluginSpecific.Options` property to access plug-in-specific options. Use the `Parameters.ModelPlugin.Base.SequenceFileName` property to determine the plug-in to which the instance data belongs. The model calls the Model Plugin –

Initialize entry point for each plug-in instance before calling the `ModelPluginOptions` callback so you can programmatically initialize or modify the plug-in instance before the model passes the instance to the client sequence file.

The `NI_DatabaseLogger` and the `NI_ReportGenerator` plug-ins also provide plug-in-specific `DatabaseOptions` and `ReportOptions` callbacks that client sequence files can override to access plug-in-specific options.

ModelPluginConfiguration

The model calls the `ModelPluginConfiguration` callback one time after calling the `ModelPluginOptions` callback for every plug-in instance. The model passes the entire set of configured plug-ins to the **ModelPluginConfiguration** parameter. Iterate through the `Parameters.ModelPluginConfiguration.Plugins` array to access each configured plug-in.

You can replace or reorder elements of the `Parameters.ModelPluginConfiguration.Plugins` array. You must ensure that the

`<NI_ModelPlugin>.Base.RuntimeVariables.InitializationState` property is set to zero for each plug-in you add to the array. After the `ModelPluginConfiguration` callback completes, the process model calls the `Model Plugin – Initialize` entry point and the `ModelPluginOptions` callback for each plug-in in the array with a

`<NI_ModelPlugin>.Base.RuntimeVariables.InitializationState` property set to zero.

Related concepts:

- [Model Plugin – Initialize](#)

Invoking Model Callbacks

Process models define model callbacks that client sequence files can override, such as the `PreUUT` or `TestReport` callbacks. Because a model plug-in is a model file, it can also define model callbacks that client sequence files can override.

The model plug-in sequence file must include the model callback sequence so the

plug-in can invoke the callback sequence. The process model must also include a copy of the callback sequence for TestStand to display the callback sequence in the Sequence File Callbacks dialog box so you can conveniently override the process model callback sequences in client sequence files.

The following table outlines ways in which you can implement callbacks and the resulting behavior for each technique. The table uses the term defined to mean that the file contains the callback sequence and the term invoked to mean that other sequences in the file call the callback sequence. If TestStand recognizes the callback at edit time, the callback appears in the Sequence File Callbacks dialog box and uses a green icon in the Sequences pane.

Description of Callback Configuration	Resulting Behavior
Defined and invoked only in the process model	Client sequence can override the callback, and the callback is recognized at edit time.
Defined and invoked only in the process model plug-in (Not Recommended)	Client sequence can override the callback, but the callback is not recognized at edit time.
Defined in the process model and the process model plug-in but invoked only in the plug-in	If the client sequence file defines the callback, the client sequence file overrides the plug-in implementation of the callback. Otherwise, TestStand invokes the plug-in definition of the callback. The process model definition of the callback does not override the plug-in definition. The callback is recognized by the client sequence file at edit time.
Defined and invoked in the process model and the process model plug-in	If the client sequence file defines the callback, the client sequence file overrides the process model and process model plug-in implementations. Sequences in the process model and the plug-in invoke the process model and plug-in instances of the callback, respectively. The process model definition of the callback does not override the plug-in definition or vice versa. The callback is recognized by the client sequence file at edit-time.

Related concepts:

- [Callback Sequences](#)

Enabling or Disabling Editing of NI_ModelPlugin Properties

Although you can provide sub-dialog boxes for plug-in end users to edit plug-in-specific option properties, plug-in end users can also use the parent configuration dialog box to edit the following properties common to all plug-ins:

- `Base.Enabled`
- `Base.DisplayNameExpression`
- `Base.NewThread`
- `Base.CompleteBeforeNextUUT`
- `PluginSpecific.Options`
- `PluginSpecific.AdditionalOptions`

You can restrict plug-in end users from editing these properties by modifying the value of the `NI.EnabledExpression` expression (string) attribute that contains a Boolean expression for the property for which you want to restrict editing. For example, the expression `CurrentUserHasPrivilege("ConfigReport")` specifies that only users with the ConfigReport user privilege can edit the property. Apply this change to the `FileGlobal` instance of the property, located within the `FileGlobals.ModelPluginComponentDescription.Default` container.

Integrating Model Plug-ins with a Custom Process Model

Process models invoke model plug-in entry points at run time by calling the sequence with the corresponding name from `ModelSupport.seq`. For example, calling the Model Plugins – Pre UUT sequence in `ModelSupport.seq` invokes the Model Plugin – Pre UUT entry point in all enabled plug-in instances. Calling the Model Plugins – Begin sequence in `ModelSupport.seq` invokes the Model Plugin – Initialize entry point and the Model Plug – Begin entry point in all enabled plug-in instances. The `ModelSupport.seq` file loads the configured plug-in instances when the process model calls the Initialize Execution Entry Point sequence.

A custom process model must call all the applicable sequences in

`ModelSupport.seq` in the following order:

1. Initialize Execution Entry Point
2. Model Plugins – Begin
3. Model Plugins – Pre Batch (Batch controller thread only)
4. Model Plugins – Batch Start (Batch controller thread only)
5. Model Plugins – Pre UUT (Test socket thread only)
6. Model Plugins – UUT Start (Test socket thread only)
7. Model Plugins – OnTheFly Step Results (Call from the `ProcessModelPostResults` callback)
8. Model Plugins – UUT Done (Test socket thread only)
9. Model Plugins – Post UUT (Test socket thread only)
10. Model Plugins – Batch Done (Batch controller thread only)
11. Model Plugins – Post Batch (Batch controller thread only)
12. Model Plugins – End



Note NI recommends that you create a custom process model by copying all the process model files from the `<TestStand>\Components\Models\TestStandModels` directory to `<TestStand Public>\Components\Models` and making changes to the copy.

A custom process model that creates a controller thread and multiple test socket threads like the Batch and Parallel process models must meet the following requirements:

All Models

- The Begin entry point in the controller thread must complete before any Begin entry point in any test socket thread starts.
- All End entry points in any test socket thread must complete before the End entry point in the controller thread starts.

Batch-Specific

- All Begin entry points in all test socket threads must complete before the Pre Batch entry point starts.
- The Pre Batch entry point must complete before any Pre UUT entry points start.

- All restarted test socket threads must reach the same sync point as the remaining, active test socket threads before any Pre UUT entry points start.
- All UUT Start entry points must complete before the Batch Done entry point starts.
- The Batch Start entry point must complete before any UUT Done entry points start.
- All Post UUT entry points must complete before the Post Batch entry point starts.

Related concepts:

- [Model Plugin – Pre UUT](#)
- [Model Plugin – Initialize](#)
- [Model Plugin – Begin](#)
- [TestStand Directory Structure](#)
- [Process Model Thread Types](#)
- [Batch Controller and Test Socket Synchronization Architecture](#)
- [Avoid Batch Synchronization Sections in Process Model Plug-in Entry Points](#)

Debugging Result Processing Plug-ins with Offline Results

You can set breakpoints in and debug model plug-ins like in any other sequence. However, if you create a result processing plug-in and the plug-in returns errors only when the TestStand Offline Results Processing Utility invokes the plug-in to process a TestStand results (`.tsr`) file, you cannot debug the plug-in from within the utility. For debugging purposes, you can process a `.tsr` file from within the TestStand sequence editor by invoking the `Process Results File [for testing and debugging]` sequence in the `NI_OfflineResultsGenerator.seq`.

The sequence processes the `.tsr` files you select with the `<Default for Offline Processing>` configuration. Edit the value of the `Locals.OfflineProcessingConfiguration` variable to specify a different configuration.

Migrating Process Model Customizations to TestStand 2012 or Later

Before you migrate customizations from TestStand 2010 or earlier into TestStand 2012 or later, NI recommends that you familiarize yourself with the process model plug-in architecture.



Note Before you customize any process model files, NI recommends that you copy the default `<TestStand>\Components\Models` directory to the corresponding `<TestStand Public>\Components\Models` directory and make modifications only to the files in the `<TestStand Public>` directory so you can easily revert unwanted changes by deleting the copy of the process model files in the `<TestStand Public>` directory.

The first step in successfully migrating existing customizations to TestStand 2012 or later is to determine where to implement changes in the newer process model files. Once you determine the new location for the customization, you must then consider how to integrate the changes with the changes present in the TestStand 2012 or later components.

In many cases, the TestStand components that contain the customizations you made also contain changes implemented in TestStand 2012 or later. This situation requires you to integrate the changes into a single file that contains the TestStand 2012 or later changes and the customizations you made. The approach you take for integrating files differs based on the following levels of complexity:

- **Changes you made to the component are less complex than changes implemented in TestStand 2012 or later**—Determine the changes you made to the component and reapply the customizations to a copy of the TestStand 2012 or later version of the component.
- **Changes you made to the component are more complex than changes implemented in TestStand 2012 or later**—Determine the changes between the TestStand 2012 or later version of the component and the original version of the component you modified and apply the TestStand 2012 or later improvements to the custom version of the component.

To facilitate this process, consider using one of the following recommended differ tools to quickly compare files.

File Type	Recommended Differ Tool
Sequence file (.seq)	TestStand File Diff and Merge utility

File Type	Recommended Differ Tool
VI (.vi)	LabVIEW Compare VIs dialog box
C code (.c, .h)	Select Edit»Diff in LabWindows/CVI
Other files (non-binary)	Third-party text differ

Categories of Customizations

Customizations typically fall into one of following categories:

- Result processing components
- Built-in components
- Custom process model components

Related concepts:

- [Process Model Plug-In Architecture](#)
- [TestStand Directory Structure](#)
- [Migrating Custom Results Processing Components](#)
- [Migrating Other Custom Built-in Process Model Components](#)
- [Migrating Custom Process Model Components](#)

Migrating Custom Results Processing Components

Many process model customizations relate to changing the way TestStand generates reports or logs data to databases. The scope of the changes determines the approach you take to implement customizations you made. You can migrate the following common types of result processing changes:

Related concepts:

- [Migrating Changes to Settings of the Default Result Processors](#)
- [Migrating Modifications to Process Model Callbacks](#)
- [Migrating Additional Results from the Client Sequence](#)
- [Migrating Modifications to Built-in Result Processors](#)
- [Migrating Custom Result Processors](#)

Migrating Changes to Settings of the Default Result Processors

The most common customizations to result processing involve modifying the configuration of the existing report generator or database logger. You typically use the Report Options or Database Options dialog boxes to implement these changes.

TestStand 2010 or earlier saves these configuration changes to disk in the `TestStandModelReportOptions.ini` and `TestStandDatabaseOptions.ini` files, respectively. TestStand 2012 or later stores the configuration data for all result processing in a single `ResultProcessing.cfg` configuration file.

Use the TestStand Migration Utility to migrate configuration data in the `.ini` files of previous versions of TestStand to the TestStand 2012 or later `.cfg` file. You can also use the utility to migrate result processing settings from TestStand 2010 or later to TestStand 2012. If you are migrating from a version of TestStand earlier than TestStand 2010, you must manually migrate the result processing settings.

Migrating Modifications to Process Model Callbacks

Important similarities and differences exist in the way you use process model callbacks in TestStand 2012 or later and in previous versions of TestStand.

Overriding Process Model Callbacks

To customize result processing for a specific type of UUT, you typically override process model callbacks in a client sequence file. This technique remains the same in TestStand 2012 or later. You can override callbacks in the same manner as in earlier versions of TestStand. As a result of this similarity, you can seamlessly migrate client sequence files that override process model callbacks from earlier versions of TestStand to TestStand 2012 or later.

Migrating Modifications to Built-in Process Model Callback Definitions

Although the process of overriding callbacks has not changed, TestStand 2012 or later changes the location where it defines the callbacks. TestStand 2012 or later defines result processing callbacks in the process model and in the result processing model plug-in, but only the process model plug-in implementation contains any steps. The callback definition in the process model exists so that the client sequence file recognizes the callback at edit time, meaning that the callback icon appears green in the Sequences pane and is present in the Sequence File Callbacks dialog box.



Note The term definition describes only callback sequences that contain the steps to implement the callback code because this is the only callback definition you should modify.

If you modified the default functionality of a results processing callback by directly modifying the callback sequence in the process model, you must copy the modifications into the callback sequence in the corresponding process model plug-in sequence file. For example, if you added steps to the TestReport callback sequence in `SequentialModel.seq` in TestStand 2010 or earlier, you must migrate these changes to the TestReport callback sequence in the `NI_ReportGenerator.seq` result processing model plug-in.

The following table shows where TestStand 2012 or later defines the TestStand 2010 SP1 or earlier results processing callback sequences:

Callback Sequence	Location of Callback Definition in TestStand 2012 or Later
TestReport	NI_ReportGenerator.seq (report generation result processing model plug-in)
BatchReport	
ModifyReportHeader	
ModifyReportEntry	
ModifyReportFooter	
ReportOptions	

Callback Sequence	Location of Callback Definition in TestStand 2012 or Later
ModifyBatchReportHeader ModifyBatchReportEntry ModifyBatchReportFooter	
DatabaseOptions LogToDatabase	<code>NI_DatabaseLogger.seq</code> (database logging result processing model plug-in)

Migrating Custom Result Processing Callbacks

If you created custom result processing callbacks in TestStand 2010 or earlier, you must migrate these callbacks and any steps that call the callbacks to the process model plug-in sequence file. To invoke the callback, create a Sequence Call step in the appropriate Model Plugin entry point. For example, if you created a custom callback to log additional information to the report immediately after TestStand generates the report, move this callback definition from the process model to the `NI_ReportGenerator` process model plug-in, then call the callback sequence in the Model Plugin — UUT Done plug-in entry point. Maintain a blank definition of the callback in the process model itself so client sequence file developers know that the callback is available to override.

Related concepts:

- [Model Plug-in Entry Points](#)
- [Model Plugin – UUT Done](#)

Migrating Additional Results from the Client Sequence

As in previous versions of TestStand, you can use the following techniques in TestStand 2012 or later to modify the data to include in the report or database:

- Directly modifying `ResultList`
- Using the `Execution.AddExtraResult` method

If you implemented these techniques in the client sequence file, you can use the client sequence file directly in TestStand 2012 or later without modification. If you implemented these techniques in the process model components in TestStand 2010 or earlier, you cannot directly migrate the component to TestStand 2012 or later.

Related concepts:

- [Result Collection](#)
- [Migrating Modifications to Built-in Result Processors](#)

Migrating Modifications to Built-in Result Processors

For more complex reporting customizations, you must often modify the underlying result processing code the process model calls. The format of results you generate determine in which process model components the modifications reside. Although the TestStand 2012 or later process models use the same underlying components in many cases, most TestStand 2012 or later components include modifications to some extent with respect to the corresponding TestStand 2010 SP1 or earlier components. The TestStand 2012 or later process models depend on many of these changes to function properly.

Refer to the following table for more information about which components to migrate for common types of customizations and the complexity of changes for each component for TestStand 2012 or later. All files mentioned in the table are located in the `<TestStand>\Components\Models\TestStandModels` directory unless otherwise specified.

Table 35.

Description of Customization	Likely Location of Changes in TestStand Components	Details on Changes in TestStand 2012 or Later
Changing the	Report style sheets:	Significant changes exist in the TestStand 2012 or

Description of Customization	Likely Location of Changes in TestStand Components	Details on Changes in TestStand 2012 or Later
appearance or formatting of the ATML or XML report	• ATML—TRML.xml • XML—report.xml, expand.xml, horizontal.xml	later versions of the ATML report style sheets. TestStand 2013 or later supports the approved version 6.01 of the ATML Test Results and Session Information schema the ATML standard defines. ATML reports that validate against version 6.01 of the schema can use the TR6_report.xml, TR6_Horizontal.xml, or TR6_Expand.xml style sheets. TestStand 2012 or later supports the unapproved version 5.0 of the ATML Test Results and Session Information schema the ATML standard defines. ATML reports that validate against version 5.0 of the schema can use the TR5_report.xml, TR5_Horizontal.xml, or TR5_Expand.xml style sheets. You can also use the ATML 2.02 standard in TestStand 2010 or later with the TRML.xml style sheet or TR_horizontal.xml style sheet.  Note No plans exist to approve version 5.0 of the ATML Test Results and Session Information schema.
Changing the appearance or formatting of the report (text, HTML)	—	Refer to the Logging additional information to the report body rows for more information about these types of changes.
Logging additional information to the report header	AddReportHeader sequence in ReportGen_<format>.seq	Minor changes exist in the TestStand 2012 or later version of these components.
Logging additional information to the report footer	AddReportFooter sequence in ReportGen_<format>.seq	No changes exist in the TestStand 2012 or later version of these components.
Logging	AddReportBody sequence in	Moderate changes exist in the TestStand 2012 or

Description of Customization	Likely Location of Changes in TestStand Components	Details on Changes in TestStand 2012 or Later
additional information to the report body (On-the-Fly disabled)	ReportGen_<format>.seq	later version of these components. • ATML—The ATMLSupport.dll file generates the report body. TestStand includes the source code for this DLL. • XML—The XML generation process is built into the TestStand Engine and is not customizable.
Logging additional information to the report body (On-the-Fly enabled)	InitializeOTF and ProcessOTFStepResult sequences in ReportGen_<format>.seq	TestStand 2012 or later includes the ProcessModelPostResults Engine callback to handle on-the-fly results. This callback executes at various intervals as a result of the on-the-fly configuration settings. Because of this change, the on-the-fly result processor must handle multiple results. The functionality in the TestStand 2010 on-the-fly results generator was heavily modified to implement this change. TestStand 2012 or later implements the InitializeOTFReport and ProcessOTFStepResults sequences in ReportGen_<format>.seq for on-the-fly reporting.

Related concepts:

- [Migrating Process Model Customizations to TestStand 2012 or Later](#)
- [TestStand Directory Structure](#)
- [ATML Test Results Reports](#)
- [Process Model Support Files](#)
- [XML Reports](#)

Migrating Custom Result Processors

For applications with specific result processing requirements, you can implement a customized result processing model plug-in. Because the TestStand 2010 or earlier process models did not support process model plug-ins, creating a custom result processor was a difficult process. Using TestStand 2010 or earlier to accomplish this

task required you to implement the result processing code and correctly integrate it into the built-in process models. The TestStand 2012 or later process model plug-in architecture decouples custom result processing code from the process model so you can focus on the result processing code itself rather than integration with the process model. The process model plug-in architecture includes the following benefits:

- Process model plug-in sequences automatically populate with process model data, which avoids the need for you to create parameters in each calling sequence to pass data
- With process model plug-ins, you can implement functionality at any point in an execution without modifying the process model files
- You can migrate process model plug-ins to future versions of TestStand with little to no modification

To migrate custom result processing changes to TestStand 2012 or later, create a new result processing model plug-in that contains the same functionality. To ease development, you can use the Result Processing dialog box to create a template process model plug-in that contains all the necessary entry points and data types.

After you create the process model plug-in, use the following table as a guide to migrate functionality into the correct process model plug-in sequences.

Type of Functionality	Likely Location in TestStand 2010 or Earlier Process Model Files	Corresponding Process Model Plug-in Sequence
Configuration dialog box for custom options	Configuration entry point in model sequence	Model Plugin — Configure Standard Options
Initialization (one time)	• ProcessSetup Callback • PreUUTLoop Callback	Model Plugin — Begin
Initialization (per UUT)	PreUUT Callback	Model Plugin — UUT Start
Initialization (per batch)	PreBatch Callback	Model Plugin — Batch Start
Processing results (per UUT)	• TestReport Callback • LogToDatabase Callback	Model Plugin — UUT Done
Processing results (per batch)	BatchReport Callback	Model Plugin — Batch Done

Type of Functionality	Likely Location in TestStand 2010 or Earlier Process Model Files	Corresponding Process Model Plug-in Sequence
batch)		
Processing results (On-the-Fly)	• ProcessModelPostResultList • Entry Engine Callback	Model Plugin — OnTheFly Step Results
Clean-up (one time)	• ProcessCleanup Callback • PostUUTLoop Callback	Model Plugin — End
Clean-up (per UUT)	PostUUT Callback	Model Plugin — UUT Done
Clean-up (per batch)	PostBatch Callback	Model Plugin — Batch Done

In addition to the process model plug-in sequences, the model plug-in template contains data types you can use to store configuration or other data relevant to the process model plug-in. Add any properties specific to your custom result processing functionality to the plug-in Options data type to store configuration options. An instance of this type exists in each process model plug-in entry point within the `PlugInSpecific` container. Use a code module in the Model Plugin — Configure Standard Options sequence to provide an interface for end users to set values for the properties you added.

TestStand stores this model plug-in specific data in a configuration file and automatically passes the data into the process model plug-in sequences when the process model calls the model plug-in sequence file.

Related concepts:

- [Creating Process Model Plug-ins](#)
- [Model Plug-in Entry Points](#)
- [Model Plugin – Configure Standard Options](#)
- [Model Plugin – Begin](#)
- [Model Plugin – UUT Start](#)
- [Model Plugin – Batch Start](#)
- [Model Plugin – UUT Done](#)
- [Model Plugin – Batch Done](#)

- [Model Plugin – OnTheFly Step Results](#)
- [Model Plugin – End](#)
- [Configuring Process Model Plug-ins](#)

Migrating Other Custom Built-in Process Model Components

Because the majority of changes in the TestStand 2012 or later process models relate to the process model plug-in architecture, the following components of the process models remain largely unchanged from the TestStand 2010 or earlier:

Related concepts:

- [Migrating Customizations to Process Model Callbacks](#)
- [Migrating Customizations to Process Model Dialog Boxes](#)
- [Migrating Error Handling Customizations](#)

Migrating Model Options Files

The model options files have the following migration actions:

- **Add configuration**—Creates a new configuration in a configuration set called `Migrated from TestStand <Version>`.



Note The Migration Utility automatically selects a unique name for the configuration if the tool has already used the default name in a previous completed migration. Refer to the migration report for the specific configuration names the tool uses.

- **Overwrite legacy options**—Copies the options files from TestStand 2010 SP1 or earlier to the current TestStand directory so you can use the previous configurations with the equivalent legacy TestStand 2010 process models in the current version of TestStand. When you migrate database options, the Migration Utility renames `TestStandDatabaseOptions.ini` to `TestStandDatabaseSchemas.ini`.
- **Both**—Adds a new configuration in the `Migrated from TestStand <Version>` configuration set and also copies the options files from a previous version of TestStand.

Migrating Customizations to Process Model Callbacks

If you made changes to callback definitions in TestStand 2010 or earlier, you can use the following techniques for migrating custom code:

- Modifying the callbacks in the process model file itself
 - Use this approach when you want to continue to use the default process model functionality.
 - You will have to migrate changes when you upgrade TestStand in the future.
- Implementing the custom callback in a process model plug-in add-on
 - Use this approach when you made complex callback modifications or if you completely rewrote the callbacks.
 - You can easily migrate changes to future versions of TestStand because you do not make any changes to the process model.

The following table summarizes the recommended approach for migrating changes to callbacks.

Callback Name	Recommendation
PreUUT	If you created a custom dialog box, migrate the callback code to a custom process model plug-in add-on. Add a Sequence Call step in the correct process model plug-in sequence to call the new callback. To suppress the default callback code in the TestStand 2012 or later process model, set the element that corresponds to the callback in the <code>Locals.ModelPluginConfiguration.RuntimeVariables.ModelDialogsEnabled</code> container to <code>False</code> in the Begin sequence of the process model plug-in. You can migrate changes to a modified version of the default dialog box. If you are using the default dialog box, you can integrate changes with the TestStand 2012 or later process model files.
PostUUT	
PreBatch	
PostBatch	
PreUUTLoop	Because these callbacks are empty by default, migrate the callback code to a custom process model plug-in. Add a Sequence Call step in the correct process model plug-in sequence to call the new callback.
PostUUTLoop	
PreBatchLoop	
PostBatchLoop	

Callback Name	Recommendation
ProcessSetup ProcessCleanup	
ModelOptions	Because the ModelOptions callback must have access to the process model properties, migrate changes to this callback to the ModelOptions callback in the <code>ModelSupport.seq</code> sequence.
Custom Callbacks	Migrate the custom code to a custom process model plug-in and add a Sequence Call step in the correct process model plug-in sequence to call the new callback. Create an empty definition for the custom callback in the process model file so that the client sequence file recognizes the custom callback at edit time, meaning that the callback icon appears green in the Sequences pane and is present in the Sequence File Callbacks dialog box.

Related concepts:

- [Migrating Custom Result Processors](#)
- [Migrating Customizations to Process Model Dialog Boxes](#)
- [Migrating Process Model Customizations to TestStand 2012 or Later](#)
- [Migrating Modifications to Process Model Callbacks](#)

Migrating Customizations to Process Model Dialog Boxes

The default TestStand process models launch dialog boxes to indicate the test status and to gather information from end users. The following table provides information about changes to each dialog box. The `modelsupport2.dll`, located by default at `<TestStand>\Components\Models\TestStandModels`, includes all the code for these dialog boxes.

If changes exist in the TestStand 2012 or later implementation of the dialog box, you can use a file differencing tool to check the differences and then integrate the changes into the customized code. Additionally, ensure that you build the DLL against the TestStand 2012 or later API. If you directly migrate a DLL from an earlier version of TestStand, rebuild the DLL against the TestStand 2012 or later API before you implement the DLL with TestStand 2012 or later.

Customized Model Dialog Box	Location of Modified Code	Notes
UUT Serial Number dialog box (Sequential process model)	modelsupport2.dll (uutdlg.c)	No changes exist in the TestStand 2012 or later version
UUT Status dialog box (Parallel process model)	modelsupport2.dll (paralleluutdlg.c)	Minor changes exist in the TestStand 2012 or later version
UUT Status dialog box (Batch process model)	modelsupport2.dll (batchuutdlg.c)	Minor changes exist in the TestStand 2012 or later version
UUT Result banner	modelsupport2.dll (banners.c)	Minor changes exist in the TestStand 2012 or later version

If you implement custom dialog boxes that your custom process model calls, you can migrate the callbacks that implement these dialog boxes.

Related concepts:

- [TestStand Directory Structure](#)
- [Migrating Customizations to Process Model Callbacks](#)

Migrating Error Handling Customizations

The following table outlines common error handling customization techniques and the recommended approach for migrating these changes to TestStand 2012 or later.

Custom Error Handling Technique	Recommended Approach for Migration to TestStand 2012 or Later
Configuring the error handling behavior in the station options	Use the TestStand Migration Utility
Using the ProcessModelPostStepRuntimeError Engine callback in the process model	This callback has no changes in TestStand 2012 or later, and you can migrate any code in this callback directly to the TestStand 2012 or later process models

Migrating Custom Process Model Components

In addition to modifying the existing components of the process model, you might have added completely new functionality to the process models. When migrating these types of customizations, consider creating a new process model plug-in to implement the same functionality. If it is possible to implement custom components as a process model plug-in, NI recommends you do so because it can greatly simplify migration of custom components in versions of TestStand later than 2012.

If you want to manage the configuration of the process model plug-in using a method other than the existing Results Processing dialog box, change the process model plug-in category. The Result Processing dialog box displays only process model plug-ins with a category (`ModelPlugin.CategorySpecific.Name`) set to `ResultProcessor`.

If you want the new process model plug-in to always execute independently of any configuration settings, configure the model plug-in as a plug-in add-on by storing the process model plug-in in the `<TestStand Public>\Components\Models\ModelPlugins\Addons` directory. Refer to the Dynamic Client Sequence File example in TestStand 2012 or later for an example of using a process model add-on.

Related concepts:

- [Creating Process Model Plug-ins](#)
- [TestStand Directory Structure](#)
- [Dynamically Set the Client Sequence File](#)

64-bit TestStand and Migrating from 32-bit TestStand

TestStand includes 32-bit TestStand and 64-bit TestStand. You can install and run 32-bit TestStand, 64-bit TestStand, or both on a computer running the 64-bit Windows operating system. 64-bit TestStand is similar to 32-bit TestStand, and, typically, you can edit and execute sequences in either version. However, the following significant differences exist between the two versions that you must consider when you migrate from 32-bit TestStand to 64-bit TestStand or write sequences to run on 32-bit TestStand and 64-bit TestStand.

Related information:

- [Supported Software](#)

Choosing between 32-bit TestStand and 64-bit TestStand

Consider the following issues when making the decision to use 32-bit TestStand or 64-bit TestStand:

- Requirements of the test system
- TestStand system requirements and supported application development environments
- Supported TestStand features
- Driver support
- Memory requirements
- Integration with existing 32-bit sequence files
- Migration issues

Related concepts:

- [Features 64-Bit TestStand does not Support](#)
- [Driver Support for 64-bit TestStand](#)
- [64-bit Architecture Offers More Memory](#)
- [Bitness-Conditional Code for 32-bit TestStand or 64-bit TestStand](#)

Related information:

- [Supported Software](#)

Activating 32-bit TestStand and 64-bit TestStand

The TestStand Version Selector does not treat 32-bit TestStand and 64-bit TestStand as separate product versions. However, 32-bit TestStand and 64-bit TestStand use separate <Program Files>, <TestStand Public>, and <TestStand Application Data> directories and therefore use independent settings, such as station options.

Use the TestStand Version Selector to select the active TestStand version and click the **Launch 32-bit** button to launch the 32-bit TestStand Sequence Editor or click the **Launch 64-bit** button to launch the 64-bit TestStand Sequence Editor. Microsoft Windows Explorer uses the same bitness version of the sequence editor you last launched to open sequence files and other TestStand files. The TestStand File Diff and Merge utility also matches the bitness of the sequence editor you last launched.

When you make a previous TestStand version active, the TestStand Version Selector deactivates the current 32-bit TestStand and 64-bit TestStand. The **Launch 64-bit** button is dimmed when a previous version of TestStand is active.

Related concepts:

- [TestStand Directory Structure](#)

Features 64-Bit TestStand does not Support

64-bit TestStand does not support the following features:

- **IVI Step Types**—Use code modules to communicate with IVI drivers.
- **HTBasic Adapter**—HTBasic steps do not execute in 64-bit TestStand.
- **Measurement Studio C++ MFC data types**—No 64-bit version of Measurement Studio C++ MFC exists.

As a workaround, you can use these features indirectly from 64-bit TestStand by remotely executing a sequence using 32-bit TestStand. However, because this technique increases overhead and forfeits the benefits the 64-bit architecture offers, NI recommends using this technique only as a temporary solution or in situations in which these features perform only a limited role in the test system. If these features are a major or critical part of the test system, NI recommends using 32-bit TestStand.

Related concepts:

- [Remote Execution in 32-bit TestStand and 64-bit TestStand](#)

Driver Support for 64-bit TestStand

The bitness of the user mode portion of any driver TestStand loads in-process must

match the bitness of TestStand, regardless of which adapter you use to communicate with the driver. As a workaround, 64-bit TestStand can use in-process 32-bit drivers by invoking them indirectly with a remote sequence call.

Drivers you invoke out-of-process using the LabVIEW Adapter do not have to match the bitness of TestStand. TestStand can invoke LabVIEW code modules using an external (out-of-process) instance of the LabVIEW development system. In this case, the bitness of LabVIEW must match the bitness of the driver, but the bitness of LabVIEW does not have to match the bitness of TestStand. You can make cross-bitness calls using the LabVIEW Adapter only when you use the LabVIEW development system. Code modules that execute using the LabVIEW Run-Time Engine execute in-process and therefore must match the bitness of TestStand.

Similarly, the bitness of an out-of-process COM server you call using the ActiveX/COM Adapter does not have to match the bitness of TestStand.

The bitness of drivers you invoke using the LabWindows/CVI Adapter must match the bitness of TestStand even when run out-of-process using the LabWindows/CVI development environment.

If most or all of the in-process drivers a test system uses are available for only one architecture, NI recommends using the bitness of TestStand that matches the drivers. If a 64-bit version of a driver you need will be available in a reasonable amount of time, consider using 64-bit TestStand and remote sequence calls as a temporary workaround to communicate with the 32-bit drivers until the 64-bit version of the driver becomes available.

Related concepts:

- [Remote Execution in 32-bit TestStand and 64-bit TestStand](#)

64-bit Architecture Offers More Memory

64-bit architectures offer a larger address space than 32-bit architectures. 32-bit architectures use 32-bit pointers and can therefore use at most 4GB of memory but often less. By contrast, 64-bit applications can use up to 16TB (16,384GB) of memory.

In most cases, the memory available for 32-bit TestStand is more than adequate to

meet the requirements of an application. Consider using 64-bit TestStand and the additional memory a 64-bit operating system offers for applications that generate large reports or handle image data.



Note 64-bit TestStand retains some size limitations of 32-bit TestStand.

Property object arrays are limited to 2^{31} (about 2.1 billion) elements. Although you can use 64-bit TestStand to increase the memory for objects you store in the array, the maximum count of objects you can store in the array remains unchanged. Additionally, 64-bit TestStand strings are limited to 4GB because COM marshals the strings in various locations within the TestStand application.

Bitness-Conditional Code for 32-bit TestStand or 64-bit TestStand

When you cannot write bitness-independent sequence files, use the `Engine.Is64Bit` property to specify that certain steps or portions of expressions execute only in 32-bit TestStand or in 64-bit TestStand.

Typically, you must dynamically load steps that execute conditionally based on bitness. If you preload steps that depend on bitness-specific code modules, the version of the step that does not match the bitness of TestStand fails to load and returns an error even though the bitness-conditional code prevents the step from executing.

Representing Pointers in Sequences in 32-bit TestStand and 64-bit TestStand

Use the Object Reference data type to manipulate and store pointers and pointer-sized handles in TestStand sequences regardless of architecture. The Object Reference data type automatically matches the bitness of TestStand. If you use the LabWindows/CVI or C/C++ DLL Adapters, use the appropriate pointer/handle parameter type when you specify a prototype.

Using the Number data type to store pointers as 32-bit integers worked well in

previous versions of TestStand because the default representation of the Number data type supports automatic conversion to and from 32-bit integer types. If you use the Number data type to represent a 64-bit pointer, the Number must use a signed or unsigned 64-bit integer representation. Storing 32-bit pointers in 64-bit integers does not work well in TestStand because you cannot automatically convert Numbers with 64-bit integer representations to and from 32-bit integer types. NI does not recommend using a numeric data type for storing pointers.

Use the pointer/handle parameter type and the constant Object Reference `Nothing` instead of a numeric parameter type and the constant Number `0` to represent NULL pointers.

Representing Pointer-Sized Numbers in Sequences in 32-bit TestStand and 64-bit TestStand

Avoid using pointer-sized integers in sequences because no equivalent TestStand numeric data type exists. An example of a pointer-sized integer is the `size_t` type in C/C++, which is the size of a pointer but is intended to be used as a true integer data type.

The TestStand Number data type supports the following representations:

- 64-bit double-precision, floating-point (default representation)
- 64-bit signed integer
- 64-bit unsigned integer

Use the default 64-bit double-precision, floating-point representation to store signed and unsigned 32-bit integers because the 64-bit double-precision, floating-point representation can represent all possible 32-bit integer values. For 32-bit integers, the default 64-bit double-precision, floating-point representation is more appropriate than using the 64-bit signed or unsigned integer representations because TestStand supports automatic conversion between the default representation and 32-bit integer types in module adapter steps. A 64-bit double-precision, floating-point representation cannot exactly represent all possible 64-bit integer values, so TestStand does not allow automatic conversion between Numbers with the 64-bit double-precision, floating-point representation and 64-bit integer data types.

If no workaround exists to using pointer-sized integers in TestStand, use one of the

following strategies:

- Use an Object Reference data type to store the number as a pointer.
- Use a TestStand Number data type with representation based on bitness.
- Use two separate variables.

Related concepts:

- [Using an Object Reference to Represent Pointer-Sized Numbers in Sequences in 32-bit TestStand and 64-bit TestStand](#)
- [Using a TestStand Number with Representation Based on Bitness to Represent Pointer-Sized Numbers in Sequences in 32-bit TestStand and 64-bit TestStand](#)
- [Using Two Separate Variables to Represent Pointer-Sized Numbers in Sequences in 32-bit TestStand and 64-bit TestStand](#)

Using an Object Reference to Represent Pointer-Sized Numbers in Sequences in 32-bit TestStand and 64-bit TestStand

If TestStand obtains a pointer-sized number from a code module and only holds or passes the value but does not mathematically manipulate the value, use a TestStand Object Reference to store the number. Using this approach avoids the need for bitness-conditional code, but you cannot access and operate on the underlying numeric value within TestStand.

Related concepts:

- [Representing Pointers in Sequences in 32-bit TestStand and 64-bit TestStand](#)

Using a TestStand Number with Representation Based on Bitness to Represent Pointer-Sized Numbers in Sequences in 32-bit TestStand and 64-bit TestStand

Use bitness-conditional code to change the representation of a number based on the bitness of TestStand. For example, if `Locals.NumVal` must be a pointer-sized number, use the following expression to set the representation appropriately based on the bitness of TestStand:


```
Locals.NumVal.Type.Representation = RunState.Engine.Is64Bit
? PropertyRepresentation_Int64 :
PropertyRepresentation_Float64
```

Related concepts:

- [Bitness-Conditional Code for 32-bit TestStand or 64-bit TestStand](#)

Using Two Separate Variables to Represent Pointer-Sized Numbers in Sequences in 32-bit TestStand and 64-bit TestStand

When you cannot change the representation of a variable, such as in a type instance, create one instance of the variable for 32-bit TestStand and another instance of the variable for 64-bit TestStand. Use bitness-conditional code to access the appropriate variable.

Locating the Correct Code Module in 32-bit TestStand and 64-bit TestStand

If a single code module cannot simultaneously support 32-bit applications and 64-bit applications, you can use TestStand search directories or the `$(Platform)` path macro to associate a step or sequence with the 32-bit version and the 64-bit version of a code module.

Related concepts:

- [Using TestStand Search Directories to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand](#)
- [Using the `\$\(Platform\)` Path Macro to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand](#)

Using TestStand Search Directories to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand

By default, 32-bit TestStand and 64-bit TestStand do not share search directories. You can associate bitness-specific versions of a code module with a sequence or step type

by placing the code module in a non-shared TestStand search directory, such as the `<TestStand Public>` directory. The TestStand `<TestStand Public>` directories use a (32-bit) or (64-bit) suffix to indicate the bitness of TestStand to which they belong. Search directories located under the `<TestStand>` installation directory, such as the `<TestStand>\Components` directory, are naturally separate because Microsoft Windows uses separate `Program Files` directories for 32-bit applications and 64-bit applications on 64-bit Windows, typically `C:\Program Files (x86)` for 32-bit applications and `C:\Program Files` for 64-bit applications.

When you migrate code modules installed in a non-shared search directory, such as the `<TestStand Public>` directory, from 32-bit TestStand to 64-bit TestStand, complete the following steps.

1. Create a 64-bit search directory structure that matches the existing 32-bit search directory structure.
2. Install the 64-bit compatible versions of each code module in the 64-bit directory in locations that match the 32-bit versions of the code module.

The existing 32-bit code modules and directory structure do not require changes. 32-bit TestStand finds the existing 32-bit code modules in the original locations.

Additionally, 32-bit TestStand and 64-bit TestStand cannot share platform-independent code modules, such as AnyCPU .NET DLLs, you reference in a non-shared search directory. You need two copies of the code module in this case to simultaneously support 32-bit TestStand and 64-bit TestStand. If you need to simultaneously support 32-bit TestStand and 64-bit TestStand, NI recommends moving platform-independent code modules to a location 32-bit TestStand and 64-bit TestStand share so you can maintain a single copy of the code module.

If a Microsoft Visual Studio, LabWindows/CVI, or other development environment project resides in a search directory for which you enabled the Search Subdirectories option in the Edit Search Directories dialog box, ensure that the project does not create intermediate or output files from a 64-bit build in the 32-bit search directory or vice versa. When searching subdirectories, 32-bit TestStand might find the 64-bit version of the DLL code module first. When this happens, the 64-bit version of the DLL hides the 32-bit version of the DLL from 32-bit TestStand, causing steps that use the code module to fail because 32-bit TestStand cannot load the 64-bit DLL. Similarly,

when searching subdirectories, 64-bit TestStand might find the 32-bit version of the DLL code module first. When this happens, the 32-bit version of the DLL hides the 64-bit version of the DLL from 64-bit TestStand, causing steps that use the code module to fail because 64-bit TestStand cannot load the 32-bit DLL.

Related concepts:

- [TestStand Directory Structure](#)

Using the \$(Platform) Path Macro to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand

You can use the `$(Platform)` path macro to simultaneously associate a step or sequence with the 32-bit version and the 64-bit version of a code module. 32-bit TestStand expands the `$(Platform)` macro to `win32`, and 64-bit TestStand expands the macro to `x64`. All adapters that require paths to code modules support the `$(Platform)` macro.

The TestStand Engine API supports the `$(Platform)` macro in the following methods:

- `Engine.FindFileEx`
- `Engine.FindPath`
- `Engine.ExpandPathMacros`

The `Engine.FindFileEx` and `Engine.FindPath` methods expand the `$(Platform)` macro automatically when searching for files or paths. The `Engine.ExpandPathMacros` method evaluates and expands the `$(Platform)` macro in a path string to provide lower-level access to the macro functionality when needed.

When a path name includes the `$(Platform)` macro, TestStand ignores the Search Subdirectories setting in the Edit Search Directories dialog box to avoid the conflicts that would otherwise arise when multiple copies of a file with the same name reside under the same root directory.

Related concepts:

- [Adapter and Code Module Support for 64-bit TestStand](#)

Adapter and Code Module Support for 64-bit TestStand

Code modules that run in-process must match the bitness of TestStand or be bitness independent, such as an AnyCPU .NET assembly. The following table provides suggested strategies for migrating from 32-bit TestStand to 64-bit TestStand or simultaneously supporting 32-bit TestStand and 64-bit TestStand.

Adapter	Code Module Type	Notes
C/C++ DLL Adapter	C/C++ DLL	• DLL must match bitness of TestStand. • Use the <code>\$(Platform)</code> path macro, a custom search directory that differs based on architecture, or the <code><TestStand Public></code> directory to locate the correct DLL.
LabWindows/CVI Adapter	C DLL	• DLL must match bitness of TestStand. • Use the <code>\$(Platform)</code> path macro, a custom search directory that differs based on architecture, or the <code><TestStand Public></code> directory to locate the correct DLL.
.NET Adapter	.NET assembly	• Use AnyCPU assemblies when possible. • Assemblies that target a particular architecture must match the bitness of TestStand. • If you use platform-specific assemblies, use the <code>\$(Platform)</code> path macro, a custom search directory that differs based on architecture, or the <code><TestStand Public></code> directory to locate the correct assembly.  Note In TestStand 2023 Q4 and earlier, you can also use the GAC to locate the correct assembly.
ActiveX/COM Adapter	In-process COM	• Server must match the bitness of TestStand.

Adapter	Code Module Type	Notes
	server	TestStand automatically locates the server that matches its bitness using Microsoft Windows registry redirection.
ActiveX/COM Adapter	Out-of-process COM server	Server does not have to match the bitness of TestStand, but TestStand uses the version that matches its bitness if that version exists.
LabVIEW Adapter	VI	- Use source-only VIs for all LabVIEW code modules to make the VIs bitness independent in most sequence files. - Bitness of VIs you run in-process with the LabVIEW Run-Time Engine must match the bitness of TestStand. - Bitness of VIs you run out-of-process does not need to match the bitness of TestStand. - Packed project libraries must match the bitness of the LabVIEW server you are using.
Sequence Call Adapter	TestStand sequence files	NI recommends using bitness-independent sequences.
None Adapter	N/A	Works the same in 32-bit TestStand and in 64-bit TestStand.
HTBasic Adapter	HTBasic .prg file	64-bit TestStand does not support HTBasic steps.
LabVIEW NXG Adapter	N/A	LabVIEW NXG modules can only be used with 64-bit TestStand.
Python Adapter	Python code module	Python version specified for use in execution must match bitness of TestStand.

Related concepts:

- [C/C++ DLL and LabWindows/CVI DLL Support for 64-bit TestStand](#)
- [TestStand Directory Structure](#)
- [.NET Code Module Support for 64-bit TestStand](#)
- [In-Process COM Servers Support in 32-bit TestStand and 64-bit TestStand](#)
- [Out-of-Process COM Servers in 32-bit TestStand and 64-bit TestStand](#)
- [LabVIEW Code Module Support for 64-bit TestStand](#)
- [Sequence Call Adapter Support for 64-bit TestStand](#)
- [Python Code Module Support for 64-bit TestStand](#)

Sequence Call Adapter Support for 64-bit TestStand

When you migrate Sequence Call Adapter steps from 32-bit TestStand to 64-bit TestStand, you typically do not have to provide separate sequence files for each architecture because you can create bitness-independent sequences. However, for sequence files that reside in non-shared search directories, you must move the sequence files to a shared location before migrating, or you must copy the sequence files to the equivalent 64-bit location. The Sequence Call Adapter supports the `$(Platform)` path macro, and you can use the `Engine.Is64Bit` property in preconditions to write bitness-conditional code. You can also use the `Engine.Is64Bit` property in an expression to specify the path to a sequence file.

Additionally, when you migrate from 32-bit TestStand to 64-bit TestStand, you must consider the behavior of the CPU affinity and the remote execution settings for Sequence Call steps.

Related concepts:

- [Using TestStand Search Directories to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand](#)
- [Using the \\$\(Platform\) Path Macro to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand](#)
- [CPU Affinity Mask Support for 64-bit TestStand](#)
- [Remote Execution in 32-bit TestStand and 64-bit TestStand](#)

CPU Affinity Mask Support for 64-bit TestStand

Setting the CPU affinity is optional. When you select **Use Custom CPU Affinity** as the CPU Affinity option for threads or executions with Sequence Call Adapter steps, you

can specify the CPU affinity mask for a new thread or new execution. Because 32-bit Microsoft Windows uses a 32-bit CPU affinity mask, and 64-bit Windows uses a 64-bit mask, the mask behaves as a pointer-sized integer. 32-bit TestStand expects the affinity mask to be a TestStand Number data type with the default (double) representation. 64-bit TestStand expects the affinity mask to be a Number data type with the 64-bit unsigned integer representation.

If a sequence needs to support only one architecture, use the representation that matches the required architecture. If the sequence must support the 32-bit architecture and the 64-bit architecture, use bitness-conditional code in the mask expression. For example, to enable all CPUs, use the following expression:

```
RunState.Engine.Is64Bit? -lui64 : -1
```

where `-1` indicates to set all bits and therefore use all CPUs.

Only the Use Custom CPU Affinity option requires specifying the mask. The Use Station Option for CPU Affinity, Use CPU Affinity of Caller, and Use All CPUs options do not require or allow specification of the affinity mask and therefore do not require bitness-conditional code.

In TestStand 2014 or later, the Default CPU Affinity for Threads option on the Preferences tab of the Station Options dialog box is a 64-bit unsigned integer for 32-bit TestStand and 64-bit TestStand. However, because of the 32-bit limitation on affinity masks in 32-bit Windows, 32-bit TestStand cannot specify affinity for processors beyond the first 32 processors if more than 32 processors are present.

When you use the TestStand Engine API to set the default CPU affinity mask, use the `StationOptions.DefaultCPUAffinityForThreadsEx` property for 32-bit applications and for 64-bit applications instead of the now obsolete `StationOptions.DefaultCPUAffinityForThreads` property.

Remote Execution in 32-bit TestStand and 64-bit TestStand

When you select **Use Remote Computer** in **Execution Options** on the Sequence Call Module tab or **Run Sequence on Remote Computer** in the **Multithreading and Remote Execution** option in the Edit Sequence Call dialog box of a TestStand User Interface to execute a sequence remotely, you can select the bitness of the TestStand remote

engine to use. By default, TestStand connects to the remote engine with the matching bitness if available on the specified host. To specify the bitness of the TestStand remote engine to use on the remote computer, use a `32\\` prefix to use the 32-bit TestStand Engine or use a `64\\` prefix to use the 64-bit TestStand Engine in the **Remote Host** control of the Sequence Call Advanced Settings dialog box or in the **Remote Host** control in the Remote Execution Settings dialog box of a TestStand User Interface.

You can use remote sequence execution as a workaround to use 32-bit TestStand features that 64-bit TestStand does not support. For example, 64-bit TestStand can use 32-bit TestStand to execute a sequence that uses the HTBasic Adapter. Similarly, you can specify that 64-bit TestStand use 32-bit TestStand to execute sequences that invoke 32-bit code modules or vice versa. Complete the following steps to call into a specific bitness of TestStand on the same computer.

1. Enable the **Allow Sequence Calls from Remote Computer to Run on this Computer** option on the Remote Execution tab of the Station Options dialog box.
2. Specify `32\\localhost` or `64\\localhost` for the remote host setting of the sequence call.



Note (64-bit TestStand) The HTBasic Adapter is not supported.

.NET Code Module Support for 64-bit TestStand

When you migrate .NET code modules from 32-bit TestStand to 64-bit TestStand, NI recommends using platform-independent AnyCPU assemblies in shared locations. Use one of the following techniques to migrate platform-specific assemblies:

- Use the `$(Platform)` path macro.
- Place the assembly in a non-shared search directory.



Note If you are using TestStand 2023 Q4 or earlier, you can also register the assembly in the GAC or use .NET remoting to call a non-matching bitness version of the DLL to migrate platform-specific assemblies.

Related concepts:

- [AnyCPU Assemblies in 32-bit TestStand and 64-bit TestStand](#)
- [Platform-Specific Assemblies in 32-bit TestStand and 64-bit TestStand](#)

AnyCPU Assemblies in 32-bit TestStand and 64-bit TestStand

AnyCPU .NET assemblies are bitness independent. You can load the same version of the assembly in 32-bit TestStand and in 64-bit TestStand. When you migrate from 32-bit TestStand to 64-bit TestStand, you do not need to modify a sequence file that relies only on AnyCPU .NET assemblies that reside in locations 32-bit TestStand and 64-bit TestStand share.

You can migrate AnyCPU assemblies located in the `<TestStand Public>` directory or other non-shared search directory using one of the following techniques:

- Copy the assembly to a shared location and reference the assembly from the new location before you migrate from 32-bit TestStand to 64-bit TestStand.
- Copy the assemblies to the equivalent 64-bit location to use in 64-bit TestStand.

NI recommends placing AnyCPU assemblies in a location 32-bit TestStand and 64-bit TestStand share so that 32-bit TestStand and 64-bit TestStand can share a single copy of each assembly. NI also recommends replacing existing 32-bit (x86) .NET assemblies with AnyCPU versions in-place before migrating from 32-bit TestStand to 64-bit TestStand.

Related concepts:

- [TestStand Directory Structure](#)
- [Using TestStand Search Directories to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand](#)

Platform-Specific Assemblies in 32-bit TestStand and 64-bit TestStand

Complete the following steps to migrate platform-specific .NET assemblies when you no longer intend to support the 32-bit architecture.

1. Replace existing 32-bit .NET code modules with 64-bit versions in-place.
2. Move the assemblies located in the `<TestStand Public>` or other non-shared search directory to the equivalent 64-bit location.

If the 64-bit version of the interface remains unchanged, TestStand sequences that depend on the code modules execute in 64-bit TestStand but no longer execute in 32-bit TestStand.

If you need to simultaneously support both architectures with a .NET code module but cannot use an AnyCPU assembly, create a 32-bit version of the assembly and a 64-bit version of the assembly and use one of the following techniques to associate the assembly with the adapter:

- Use the `$(Platform)` path macro. Although the default 32-bit subdirectory for .NET projects is `x86` in Microsoft Visual Studio, TestStand expands `$(Platform)` to `win32` for the .NET Adapter just as it does for the C/C++ DLL and LabWindows/CVI Adapters.
- Use non-shared TestStand search directories.
- Write bitness-conditional code when the 32-bit version and the 64-bit version of a .NET code module use different interfaces.



Note If you are using TestStand 2023 Q4 or earlier, you can also register the assembly in the GAC to associate the assembly with the adapter. If you register versions of an assembly in the GAC that support the x86 (32-bit) architecture and the x64 (64-bit) architecture, the .NET Runtime Engine automatically loads the correct version of the assembly.



Note If you are using TestStand 2023 Q4 or earlier, you can use .NET remoting to call a non-matching bitness of a platform-specific assembly out-of-process.

Related concepts:

- [TestStand Directory Structure](#)
- [Using the \\$\(Platform\) Path Macro to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand](#)
- [Components Directory](#)
- [Bitness-Conditional Code for DLLs with Different 32-bit Interfaces and 64-bit Interfaces](#)

ActiveX/COM Code Module Support for 64-bit TestStand

When you migrate ActiveX/COM code modules from 32-bit TestStand to 64-bit TestStand or need to simultaneously support 32-bit TestStand and 64-bit TestStand, you must consider how you use in-process and out-of-process COM servers.

Related concepts:

- [In-Process COM Servers Support in 32-bit TestStand and 64-bit TestStand](#)
- [Out-of-Process COM Servers in 32-bit TestStand and 64-bit TestStand](#)

In-Process COM Servers Support in 32-bit TestStand and 64-bit TestStand

In-process COM servers must match the bitness of TestStand. If both versions of the server are registered, TestStand automatically locates the correct server using Microsoft Windows registry redirection, and the ActiveX/COM Adapter step typically does not require modification. Use bitness-conditional code with the ActiveX/COM Adapter when the 32-bit version and the 64-bit version of COM interfaces differ.

Related concepts:

- [Bitness-Conditional Code for DLLs with Different 32-bit Interfaces and 64-bit Interfaces](#)

Out-of-Process COM Servers in 32-bit TestStand and 64-bit TestStand

Because COM inherently supports inter-process communication across architectures, only a single instance of an out-of-process COM server might be necessary to support the same adapter step in 32-bit TestStand and in 64-bit TestStand. By default, if only one version of the server is registered, TestStand automatically locates that version of the server. If a 32-bit version and a 64-bit version of the server are registered, 32-bit TestStand locates the 32-bit server, and 64-bit TestStand locates the 64-bit server. You can change this default behavior in the registry on a per-server basis to require matching bitness, but changing this behavior affects all clients of the server on the computer, not just TestStand. For out-of-process COM servers, TestStand relies on the launch behavior specified for the server in the Microsoft Windows Registry. There is no way within TestStand to control which bitness of an out-of-process COM server TestStand uses. The particular functionality and performance requirements of the

server and the application determine any migration tasks you must perform.

Although out-of-process servers do not have to match the bitness of TestStand, problems might arise depending on the design and expected behavior of the server when the server bitness does not match the bitness of TestStand. For example, if a 64-bit version of the server is expected to be registered but is missing, TestStand automatically locates the 32-bit server by default. In this scenario, the step might function correctly during basic testing but might have poor performance, run out of memory unexpectedly, or exhibit other subtle differences in behavior.

Another issue to consider is that when you register a 32-bit version and a 64-bit version of a singleton server, the server is actually a doubleton composed of two separate singleton servers. 64-bit clients share a single instance of the 64-bit server, and 32-bit clients share a separate single instance of the 32-bit server. The design and functionality of the server determine the issues you must resolve.

Use bitness-conditional code with the ActiveX/COM Adapter when the 32-bit version and the 64-bit version of COM interfaces differ.

Related concepts:

- [Bitness-Conditional Code for DLLs with Different 32-bit Interfaces and 64-bit Interfaces](#)

C/C++ DLL and LabWindows/CVI DLL Support for 64-bit TestStand

When you migrate C/C++ DLL and LabWindows/CVI DLLs from 32-bit TestStand to 64-bit TestStand or want to simultaneously support 32-bit TestStand and 64-bit TestStand, you must ensure that TestStand locates the correct DLL, account for DLLs with different 32-bit interfaces and 64-bit interfaces, and use the appropriate pointers and handles.

For the C/C++ DLL and LabWindows/CVI Adapters, the performance for calling 32-bit and 64-bit DLL code modules is similar. However, the performance for loading DLLs into memory is slightly slower for 64-bit DLLs than for 32-bit DLLs.

Related concepts:

- [Locating the Correct DLL in 32-bit TestStand and 64-bit TestStand](#)
- [Bitness-Conditional Code for DLLs with Different 32-bit Interfaces and 64-bit Interfaces](#)
- [Pointers and Handles in 32-bit TestStand and 64-bit TestStand](#)

Locating the Correct DLL in 32-bit TestStand and 64-bit TestStand

TestStand executes C/C++ DLL code modules in-process. Similarly, TestStand executes LabWindows/CVI code modules in-process when you use the LabWindows/CVI Run-Time Engine. Although TestStand supports executing LabWindows/CVI code modules out-of-process using the LabWindows/CVI Development Environment, TestStand requires the bitness of the external code module to match the bitness of TestStand. Therefore, you must use separate 32-bit versions and 64-bit versions of DLL code modules for the C/C++ DLL Adapter or the LabWindows/CVI Adapter to simultaneously support 32-bit architectures and 64-bit architectures.

TestStand automatically locates the correct versions of C/C++ and LabWindows/CVI code modules in non-shared search directories, such as the `<TestStand Public>` directory. If the prototypes of the functions that the 32-bit version and 64-bit version of the DLLs export are identical and each version resides under the correct search directory, TestStand sequences and step types that depend on the code modules execute in 32-bit TestStand or in 64-bit TestStand without modification. (For prototypes that change, see Bitness Conditional Code for DLLs with Different 32-bit Interfaces and 64-bit Interfaces.)

If you migrate code modules you specify with paths relative to the sequence file or another location that 32-bit TestStand and 64-bit TestStand share and you no longer intend to support the 32-bit architecture, you can replace all DLL code modules with 64-bit versions in-place. If the prototypes of the functions that the 32-bit version and the 64-bit version of the DLLs export are identical, TestStand sequences that depend on the code modules execute in 64-bit TestStand but no longer execute in 32-bit TestStand.

If you migrate code modules you specify with paths relative to the sequence file or another location that 32-bit TestStand and 64-bit TestStand share and you need to simultaneously support the 32-bit architecture and the 64-bit architecture, complete the following steps to associate the bitness-specific version of the DLL with a step.

1. Create a `win32` subdirectory and an `x64` subdirectory in the directory that contains the DLL.
2. Move the 32-bit version of the DLL to the `win32` subdirectory.
3. Place the 64-bit version of the DLL in the `x64` subdirectory.
4. Update the module path to `$(Platform)\filename.dll` in the code module specification. 32-bit TestStand converts the path to `win32\filename.dll`, and 64-bit TestStand converts the path to `x64\filename.dll`.

Regardless of whether you need to simultaneously support 32-bit TestStand and 64-bit TestStand, NI recommends using the `$(Platform)` path macro and platform subdirectories for all C/C++ and LabWindows/CVI code modules you call that reside in directories that 32-bit TestStand and 64-bit TestStand share to simplify future migration.

Related concepts:

- [Using TestStand Search Directories to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand](#)
- [TestStand Directory Structure](#)
- [Bitness-Conditional Code for DLLs with Different 32-bit Interfaces and 64-bit Interfaces](#)

Bitness-Conditional Code for DLLs with Different 32-bit Interfaces and 64-bit Interfaces

Function names and parameters can differ between the 32-bit version of an interface and the 64-bit version of an interface for a code module.

Parameter type differences might not be obvious because preprocessor macros can generate types. The C/C++ `size_t` type is a common example of a type that changes between the 32-bit version and the 64-bit version of a code module. `size_t` is a 32-bit unsigned integer in 32-bit DLLs and a 64-bit unsigned integer in 64-bit DLLs. TestStand treats 32-bit versions and 64-bit versions of prototypes that contain `size_t` parameters as distinct and cannot execute both instances from a single adapter step.

Create a 32-bit version of a step and a 64-bit version of a step and use the `Engine.Is64Bit` property to write conditional code to call code modules with dissimilar 32-bit versions and 64-bit versions from the same sequence. Use the

`!RunState.Engine.Is64Bit` precondition in the 32-bit version of the step and use the `RunState.Engine.Is64Bit` precondition in the 64-bit version of the step to execute the version of the step that matches the bitness of TestStand. Alternatively, you can use the `RunState.Engine.Is64Bit` property in an If step to achieve similar results. Set both steps to load dynamically. If you preload the steps, the version of the step that does not match the bitness of TestStand fails to load and returns an error.

Related concepts:

- [Bitness-Conditional Code for 32-bit TestStand or 64-bit TestStand](#)

Pointers and Handles in 32-bit TestStand and 64-bit TestStand

TestStand C/C++ DLL Adapter and LabWindows/CVI Adapter steps can support pointers and pointer-sized handles from a single adapter step. For example, 32-bit TestStand and 64-bit TestStand recognize a `char *` (pointer to character) as a string and a `CAObjHandle` as a LabWindows/CVI ActiveX Automation Handle. Similarly, a single step can handle functions that take values by reference.

However, if you treat the pointer as a number, a single step cannot simultaneously support the 32-bit case and the 64-bit case. For example, 32-bit TestStand supports treating LabWindows/CVI ActiveX Automation Handles as 32-bit integers, and you can specify the type `int` instead of `CAObjHandle` in the C source prototype. You cannot use this strategy for 64-bit TestStand because the `int` data type is always 32 bits. In this case, for a 64-bit version of the DLL, you would have to replace the `int` (32-bit) data type with the `long long` (64-bit) data type, which would change the signature of the function and require a separate step and conditional code to work in 32-bit TestStand and in 64-bit TestStand.

NI recommends using the `CAObjHandle` data type in C prototypes in conjunction with the TestStand Object Reference data type in the sequence when you pass LabWindows/CVI ActiveX Automation Handles as parameters in adapter steps so the same TestStand step works in the 32-bit case and in the 64-bit case.

Using numbers as pointers when you pass NULL to a function can result in incorrect behavior. In 32-bit TestStand, you can specify the type of a pointer argument as a 32-bit integer in an adapter step and pass 0. Using this technique in 64-bit TestStand

results in undefined behavior, such as data corruption or a crash. The code module expects all 64-bits of the pointer to be 0 for NULL, but 64-bit TestStand interprets the argument as only 32-bits, sets only the low order 32-bits, and leaves the high order bits undefined. In this case, to achieve correct results in 64-bit TestStand, you would have to set the type of the number in the prototype to a 64-bit integer and pass `0i64`. However, the 64-bit-specific solution would not work in 32-bit TestStand.

To pass NULL as a parameter, specify the data type as a pointer in TestStand and pass `Nothing`. NI recommends this approach in all cases, regardless of migration or cross-platform support requirements, because TestStand converts `Nothing` to the correct NULL pointer representation for the current architecture. The `Nothing` constant is an Object Reference type and is thus a more semantically accurate representation of a NULL pointer than `0` or `0i64`.

For C DLLs that do not have associated type libraries or LabWindows/CVI embedded type information, TestStand cannot detect and warn when pointers are incorrectly treated as numbers. One situation in which this issue might arise is when you call Microsoft Win32 API functions in `user32.dll` or `kernel32.dll` from the TestStand C/C++ DLL Adapter. NI recommends that when you migrate to 64-bit TestStand, you review all C/C++ DLL and LabWindows/CVI Adapter steps that use these and similar APIs for incorrect use of the number `0` as NULL for pointers.

Related concepts:

- [Representing Pointers in Sequences in 32-bit TestStand and 64-bit TestStand](#)

LabVIEW Code Module Support for 64-bit TestStand

When you migrate LabVIEW code modules from 32-bit TestStand to 64-bit TestStand, you must consider the architectures you need to support, how to handle Express VIs and property node calls, and cross-bitness support using the LabVIEW development system.

Related concepts:

- [Migrating LabVIEW Code Modules from 32-bit TestStand to 64-bit TestStand](#)
- [Simultaneously Supporting 32-bit TestStand and 64-bit TestStand](#)
- [Express VIs and Property Node Calls in 32-bit TestStand and 64-bit TestStand](#)

- [Cross-Bitness Support using the LabVIEW Development System](#)

Migrating LabVIEW Code Modules from 32-bit TestStand to 64-bit TestStand

Complete the following steps if you migrate code modules from 32-bit TestStand to 64-bit TestStand and you no longer intend to support the 32-bit architecture.

1. Mass compile all the VIs.
2. Regenerate the required packed project libraries with a supported version of 64-bit LabVIEW.
3. Replace dependent DLLs with 64-bit versions.
4. In TestStand, select **Tools»Update VI Calls** to update Express VIs and property node calls with a supported version of 64-bit LabVIEW.

Simultaneously Supporting 32-bit TestStand and 64-bit TestStand

If you need to simultaneously support the 32-bit architecture and the 64-bit architecture, use source only VIs for all LabVIEW code modules. The appropriate LabVIEW development system compiles and adds the compiled VI code to the LabVIEW object cache if the code is not already present in the cache. Subsequent calls use the code in the LabVIEW object cache.

If you are using packed project libraries or VIs that depend on DLLs, complete the following steps to associate the bitness-specific version of the packed project library or VI with a step.

1. Create a `win32` subdirectory and an `x64` subdirectory in the sequence file directory.
2. Move the bitness-specific version of the packed project library or VI into the appropriate subdirectory.
3. Update the module path to `$(Platform)\filename.lvlibp` or `$(Platform)\filename.vi`. 32-bit TestStand converts the path to `win32\filename.lvlibp` or `win32\filename.vi`, and 64-bit TestStand converts the path to `x64\filename.lvlibp` or `x64\filename.vi`.

Express VIs and Property Node Calls in 32-bit TestStand and 64-bit TestStand

LabVIEW steps that invoke Express VIs or property nodes embed the configured VI in

step properties as files that include the front panel, block diagram, and compiled code.

You can use the 32-bit LabVIEW development system and the 64-bit LabVIEW development system to execute steps with embedded VIs without modifying the sequence file because LabVIEW recompiles the embedded VIs at execution. If you use the LabVIEW Run-Time Engine (RTE) to execute steps, the bitness of TestStand must match the bitness of the LabVIEW development system that last configured or saved the embedded VI. In TestStand, select **Tools»Update VI Calls** to update Express VIs and property node calls to use the appropriate version of LabVIEW.

Using Express VIs and Property Node Calls in Step Types

When you use the LabVIEW RTE to call a LabVIEW step configured with Express VIs or property node calls, you must provide separate step types that correspond to each version and bitness of LabVIEW you support. This restriction does not apply when you use the LabVIEW development system to call step types configured with Express VIs or property node calls because LabVIEW recompiles the embedded VI. Selecting **Tools»Update VI Calls** does not update step types. You can allow users to manually update the prototype to specify the version and bitness of LabVIEW to use, but this action causes the step type to change.

Cross-Bitness Support using the LabVIEW Development System

When you use the LabVIEW development system to call VIs, you can call 32-bit VIs from 64-bit TestStand and 64-bit VIs from 32-bit TestStand. Select one of the following options in the LabVIEW Adapter Configuration dialog box to specify which LabVIEW development system version to launch:

- **Use Active LabVIEW Version**—Launches the current active 32-bit or 64-bit LabVIEW development system, with a preference to match the bitness of TestStand.
- **Use Active 32-bit Version**—Launches the current active 32-bit LabVIEW development system.
- **Use Active 64-bit Version**—Launches the current active 64-bit LabVIEW development system.

TestStand does not support cross-bitness LabVIEW calls when you configure the LabVIEW Adapter to use the LabVIEW Run-Time Engine.

Call Executable Step Type Support for 64-bit TestStand

When you migrate from 32-bit TestStand to 64-bit TestStand or when you need to simultaneously support the 32-bit architecture and the 64-bit architecture, you must consider how to set the **Process Handle Location** option on the Wait Options tab of the Call Executable edit tab in the TestStand Sequence Editor or on the Wait Options tab of the Configure Call Executable dialog box in a TestStand User Interface. The Process Handle Location option stores a process handle as a pointer-sized value. 32-bit TestStand expects this expression to be of type Object Reference or Number with the default representation (double). 64-bit TestStand expects this expression to be of type Object Reference.

Use the `Step.ProcessHandle` Call Executable step type property, which is a Number with the default (double) representation, only for backward compatibility. For new code and code that needs to be 64-bit compatible, use the `Step.ProcessHandlePtr` property instead. The `Step.ProcessHandlePtr` Call Executable step type property is an Object Reference variable that can store a pointer-sized integer regardless of the bitness of TestStand. When you pass this value into a code module, you can use the pointer/handle type for C/C++, the `IntPtr` type for the .NET Adapter, or use the expression functions `UInt64` or `Float64` to convert the value to a 64-bit number for 64-bit TestStand or to a 32-bit number for 32-bit TestStand as needed.

CPU Affinity Step Type Support for 64-bit TestStand

You can use the CPU Affinity step type to specify the CPU affinity mask of a process or thread. Because 32-bit Microsoft Windows uses a 32-bit CPU affinity mask, and 64-bit Windows uses a 64-bit mask, the mask behaves as a pointer-sized integer. 32-bit TestStand expects the affinity mask to be a TestStand Number data type with the default (double) representation. 64-bit TestStand expects the affinity mask to be a Number data type with the 64-bit unsigned integer representation.

If a sequence needs to support only one architecture, use the representation that matches the required architecture. If the sequence must support the 32-bit architecture and the 64-bit architecture, use bitness-conditional code in the mask expression. For example, to enable all CPUs, use the following expression:

```
RunState.Engine.Is64Bit? -lui64 : -1
```

where `-1` indicates to set all bits and therefore use all CPUs.

Synchronization Object Support for 64-bit TestStand

You can use a prefix to share synchronization objects between 32-bit TestStand and 64-bit TestStand. When the synchronization object name begins with an asterisk or computer name, you can use a `32` or `64` prefix to specify using 32-bit TestStand or 64-bit TestStand to host the out-of-process synchronization object. For example, the name `64*syncobj` specifies a synchronization object called `*syncobj` in the 64-bit TestStand Synchronization Server, even when used from 32-bit TestStand. If you do not use a bitness prefix, 32-bit TestStand hosts out-of-process synchronization objects in a 32-bit process, and 64-bit TestStand hosts out-of-process synchronization objects in a 64-bit process.



Note 32-bit versions of the Microsoft Windows operating system ignore the `64` prefix. If you request a 64-bit synchronization object from a 32-bit TestStand Synchronization Server running on 32-bit Windows, you will receive a 32-bit synchronization object.

Session Manager Support for 64-bit TestStand

TestStand installs and registers separate 32-bit NI Session Manager and 64-bit NI Session Manager servers on computers running 64-bit Microsoft Windows, even when you install only 32-bit TestStand. The 64-bit Session Manager works similarly to the 32-bit Session Manager. However, you cannot share session handles the out-of-process Session Manager creates between 32-bit processes and 64-bit processes because different servers generate the 32-bit session handles and 64-bit session handles. For example, if 64-bit TestStand calls a 32-bit LabVIEW VI, you cannot use session handles created in the LabVIEW VI in 64-bit TestStand or vice versa. Similarly, you cannot share handles between 32-bit TestStand and 64-bit TestStand when one instance of TestStand uses remote execution to invoke the other instance of TestStand.

Python Code Module Support for 64-bit TestStand

When you migrate Python code modules from 32-bit TestStand to 64-bit TestStand, NI recommends that you install the 64-bit CPython interpreter of the intended version in the test system. Once the 64-bit CPython interpreter is installed, you should also update the PATH environment variable with the path to the interpreter.

Deploying 32-bit TestStand and 64-bit TestStand Systems

The 32-bit TestStand Deployment Utility creates only 32-bit deployments and installers, and the 64-bit deployment utility creates only 64-bit deployments and installers. You cannot use the 64-bit installer on 32-bit operating systems.

NI recommends creating a separate deployable image for the 32-bit version of a test system and the 64-bit version of a test system. The deployment specification files (`.tsd`) the deployment utility creates are bitness independent except for installation destination issues. Therefore, you can create a single `.tsd` and use it to deploy a test system using the 32-bit deployment utility and the 64-bit deployment utility.

There are some cases in which you want to create a system that has 32-bit files and 64-bit files, such as when you create a custom sequence editor or create a library for a third party to reuse. If you want to create a deployment that contains 32-bit versions of files and 64-bit versions of files, you must add all the files in the test system to the workspace. If the deployment contains VIs, you must enable the **Include without Processing Item or Dependencies** option on the Distributed Files tab of the deployment utility for all the VIs to prevent the deployment utility from automatically compiling the VIs with the active version of LabVIEW.

When you deploy VIs, you must use a version of LabVIEW that matches the bitness of the deployment utility because the utility creates a LabVIEW distribution and automatically removes the source-only flag from the VIs.

You can use the `$(Platform)` path macro to specify paths in sequence files, and the deployment utility uses the search directories for the current bitness of the utility to locate files. When the deployment utility processes sequence files and creates the distribution image, the bitness specific directory replaces the path macro in the sequence files.

You must use the 32-bit TestStand Deployment Utility to create a patch for a 32-bit full deployment, and you must use the 64-bit TestStand Deployment Utility to create a patch for a 64-bit full deployment. Create a new full deployment to update a 32-bit test system to a 64-bit test system.

Related concepts:

- [Installer Issues for 32-bit TestStand and 64-bit TestStand](#)
- [Using the \\$\(Platform\) Path Macro to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand](#)
- [Cross-Bitness Support using the LabVIEW Development System](#)
- [Remote Execution in 32-bit TestStand and 64-bit TestStand](#)

Installer Issues for 32-bit TestStand and 64-bit TestStand

NI recommends creating a separate installer for the 32-bit version of a test system and for the 64-bit version of a test system. If you use the deployment utility to build an installer that includes the TestStand Runtime, the utility also includes dependent runtimes if any are necessary.

Installers you build with the deployment utility install files to different directories depending on the bitness of the deployment utility you used to build the installer, as shown in the following table.

Destination	32-bit Directory	64-bit Directory
TestStand directory	32-bit TestStand directory	64-bit TestStand directory
Program Files	32-bit operating system: Program Files 64-bit operating system: Program Files x86	64-bit operating system: Program Files
National Instruments directory	32-bit National Instruments directory	64-bit National Instruments directory
TestStand Public directory	32-bit TestStand Public directory	64-bit TestStand Public directory
TestStand Application Data directory	32-bit TestStand Application Data directory	64-bit TestStand Application Data directory

Destination	32-bit Directory	64-bit Directory
System WOW64 directory	N/A	<Windows>\SysWOW64



Note 64-bit operating systems store 64-bit libraries in the `System32` directory and 32-bit libraries in the `SysWOW64` directory. When you create a 64-bit deployment, install shared libraries in the `System32` directory and install 32-bit libraries in the `SysWOW64` directory. Refer to Microsoft MSDN documentation for more information about using a file system redirector.

API Updates for 64-bit TestStand

Some TestStand 2013 or earlier API methods and properties represents pointers or pointer-sized values as 32-bit integral data types, usually `long`, which does not work for 64-bit applications. The following table lists obsolete properties and methods and the corresponding TestStand properties and methods that support 32-bit applications and 64-bit applications.

Table 36. NI TestStand

Class	Obsolete (32-bit only) Property or Method	32-bit TestStand and 64-bit TestStand Property or Method
Engine	<code>LargeImageList</code>	<code>LargeImageListEx</code>
Engine	<code>RegisterUIMessageCallback</code>	<code>RegisterUIMessageCallbackEx</code>
Engine	<code>SmallImageList</code>	<code>SmallImageListEx</code>
Station Options	<code>DefaultCPUAffinityForThreads</code>	<code>DefaultCPUAffinityForThreadsEx</code>
InstrSession	<code>CloseFunction</code>	<code>CloseFunctionEx</code>
InstrSession	<code>GetHandle</code>	<code>GetHandleEx</code>

Table 37. NI Session Manager

Class	Obsolete (32-bit only) Property or Method	32-bit TestStand and 64-bit TestStand Property or Method
Engine	<code>LargeImageList</code>	<code>LargeImageListEx</code>

Class	Obsolete (32-bit only) Property or Method	32-bit TestStand and 64-bit TestStand Property or Method
Engine	RegisterUIMessageCallback	RegisterUIMessageCallbackEx
Engine	SmallImageList	SmallImageListEx
Station Options	DefaultCPUAffinityForThreads	DefaultCPUAffinityForThreadsEx
InstrSession	CloseFunction	CloseFunctionEx
InstrSession	GetHandle	GetHandleEx

32-bit TestStand supports the obsolete properties and methods for compatibility, but NI recommends using the obsolete API only when you migrate existing 32-bit TestStand systems to 32-bit TestStand. For TestStand 2014 or later applications and for updates to existing systems, use the TestStand API regardless of current requirements for 64-bit support. The obsolete properties and methods return errors when invoked in 64-bit TestStand.

In addition, the following new methods support attaching user-defined pointer-sized values to sessions and function similarly to the existing `InstrSession.AttachLong` and `InstrSession.GetLong` methods:

- `InstrSession.AttachLongPtr`
- `InstrSession.GetLongPtr`

Continue to use the `InstrSession.AttachLong` and `InstrSession.GetLong` methods with values that are 32-bits regardless of architecture.

TestStand Environments

TestStand environments enable two or more TestStand system configurations to exist side-by-side on a single system. You can specify unique application data directories and public folders for each configuration of TestStand, allowing settings, custom components, and variables normally global to the system to be maintained separately within each environment. For example, each TestStand environment can independently maintain its own station globals, type palettes, and customized process

models. It is important to note some limitations of TestStand environments:

- TestStand environments do not eliminate the requirement to version switch when changing between different versions of TestStand installed on a system.
- Environments do not provide full isolation for simultaneously executing instances of the TestStand engine. While it is possible to run multiple instances of the TestStand engine simultaneously, each in a different environment, some features of TestStand, such as out-of-process synchronization objects, are shared by all running instances.
- The TestStand remote engine does not currently support environments and therefore always runs in the global environment.

Related concepts:

- [Defining an Environment](#)
- [Environment Behavior](#)
- [Choosing an Environment](#)
- [General Environment Limitations](#)
- [Changing Environments](#)

Defining an Environment

TestStand environments are defined by an INI text file with the `.tsenv` extension. The file can specify the TestStand Public directory, TestStand Application (CommonAppData) directory (which implicitly defines the TestStand Config directory), and/or the TestStand LocalApplication (LocalAppData) directory. For example, an environment file might look like this:

```
[TestStandPaths]
```

```
CommonAppData = "EnvironmentRoot\\CommonAppData"
```

```
LocalAppData = "EnvironmentRoot\\LocalAppData"
```

```
Public = "EnvironmentRoot\\TestStandPublic"
```

The paths may be absolute or relative to the location of the environment file.



Note The TestStand Deployment Utility requires relative paths to deploy an environment.

Any directory not specified, or specified as an empty string, will map to the default location that TestStand uses when no environment is specified, also called the global environment.

You can edit `.tsenv` files with a text editor or use the Configure Environment dialog box available in the Sequence Editor under **Configure»Environment**.

To define the environment, you must define the following fields in the `.tsenv` file:

- **CommonAppData**—Specifies the `<TestStand Application Data>` directory. Specifying CommonAppData for an environment also implicitly specifies the TestStandConfig directory, which contains various TestStand configuration files. Environments with different CommonAppData directories have different configuration files, and therefore do not share the associated settings and data. For example, environments with different CommonAppData locations have independent sets of Station Globals, StationOptions, and model plugin configurations. To simplify user management, all TestStand environments share the global `Users.ini`. This means there is still only a single set of TestStand users for a given system. Use the `StationOptions.UserFilePath` property to obtain the path of the `Users.ini` file, regardless of the current environment.



Note The TestStand engine creates default versions of any required configuration files that are missing at startup. To base a new environment on an existing configuration, copy any desired configuration files from the existing environment to the configuration directory of the new environment.

- **LocalAppData**—Specifies the `<TestStand Local Application Data>` directory. This directory contains configuration files typically related to the layout and/or state of the Sequence Editor or TestStand User Interface. Data and settings persisted in these files are maintained separately from those of environments specifying a different LocalAppData. The Sequence Editor and TestStand User Interfaces create default versions of required configuration files that are missing at startup.



Note In the global environment, each Windows user account has its own LocalAppData directory. When an environment specifies the LocalAppData directory, all Windows user accounts share the specified directory.

- **Public**—Specifies the `<TestStand Public>` directory, which can contain modifications and customizations to existing TestStand components, custom user components, and other TestStand-related files. For example, specifying the public directory allows an environment to have its own customized language files, process models, and callback files.



Note If the AdapterSupport folder is missing in an environment with a custom Public directory, the TestStand Engine creates it at startup by copying the global AdapterSupport folder. No other public files are copied automatically, but you can copy them manually if required.

Related concepts:

- [<TestStand Application Data> Directory](#)
- [TestStand Local Application Data Directory](#)
- [TestStand Public Directory](#)

Environment Behavior

Environments change the locations of the TestStand Public directory, the TestStandCommonAppData directory (which implicitly defines the location of TestStand Config directory), and/or the TestStand LocalAppData directory. These settings affect the associated TestStand Search Directories, as well as the paths returned by the `Engine.GetTestStandPath` method. The table below describes how environment settings affect the values `Engine.GetTestStandPath` returns and the built-in search directory paths.

TestStand Paths Value	Environment Location	Affected Search Directory
TestStandPath_Bin	Not affected by environment	
TestStandPath_CommonAppData	CommonAppData	

TestStand Paths Value	Environment Location	Affected Search Directory
TestStandPath_Config	CommonAppData\Cfg	
TestStandPath_LocalAppData	LocalAppData	
TestStandPath_NIComponents	Not affected by environment	
TestStandPath_Public	Public	TestStand Public
TestStandPath_PublicComponents	Public\Components	TestStand Public Components
TestStandPath_Temp	Not affected by environment	
TestStandPath_Temporary	Not affected by environment	
TestStandPath_TestStand	Not affected by environment	

Environments are largely transparent to an application that uses the Engine. GetTestStandPath and Search Directories APIs to obtain directory locations. In most cases, an application should not care whether it is running within an environment or not. For applications that do require information about the environment in which they are running, the following API is available:

- **Engine.GetEnvironmentPath**—Returns the path of the current environment, or empty string if the TestStand Engine is running in the global environment.
- **Engine.GetTestStandPath**—The following constants return the associated path for the global environment: TestStandPath_GlobalConfig, TestStandPath_GlobalPublic, TestStandPath_GlobalCommonAppData, and TestStandPath_GlobalLocalAppData.

Choosing an Environment

To launch the TestStand Sequence Editor or a TestStand User Interface in an environment, specify the `/env` command line switch followed by the path of the `.tsenv` file. For example:

```
SeqEdit.exe /env "MyEnvironment.tsenv"
```

If no environment is specified, the application runs without an environment, and the TestStand Engine uses the default locations for the various directories. Running without an environment is referred to as running in the “global” environment, though

there is no global environment file.

For convenience, you can specify the global environment by explicitly passing an empty string for the environment file path on the command line. For example, both

```
SeqEdit.exe
```

and

```
SeqEdit.exe /env ""
```

launch the Sequence Editor in the global environment.

The following TestStand executables all support the `/env` command line switch to specify an environment:

- TestStand Sequence Editor
- TestStand User Interfaces
- TestStand Offline Results Processing Utility
- TestStand Sequence Analyzer Stand-Alone Application
- TestStand File Diff and Merge Utility
- Database Viewer Application

TestStand executables that launch other TestStand executables that use the TestStand Engine need to pass the current environment at launch, assuming they want the newly launched executable to use the same environment. For example, when you select the **Tools»DatabaseViewer** option in the Sequence Editor, the Sequence Editor uses `Engine.GetEnvironmentPath` with the `/env` switch to pass the current environment to `DatabaseView.exe`. To see this behavior, select **Tools»Customize** and examine the definition of Database Viewer command.



Note The Sequence Editor, full-featured TestStand user interfaces, Offline Results Processing Utility, Sequence Analyzer, and File Diff and Merge Utility all indicate the current environment in the status bar.

Related concepts:

- [Creating Custom User Interfaces](#)

General Environment Limitations

Using TestStand environments can present limitations in the following scenarios:

- **Multibyte Strings**—If the global environment has multibyte strings enabled, then all environments must have multibyte strings enabled. This is mainly because all environments share the `Users.ini` file of the global environment.
- **User Management**—Environments do not support loading an alternate set of users and passwords that differ from the global environment. All environments use the `Users.ini` file defined in the global environment, which defaults to `<TestStandApplication Data>\Cfg\Users.ini`. Nevertheless, certain forms of privilege escalation can be accomplished by creating custom configuration folders. For example, a user who cannot normally configure the engine can effectively do so within the context of an environment they create. To accomplish this, the user can create an environment with a custom `CommonAppData` directory and prepopulate the directory with configuration files.
- **Disable Environments Feature**—You can completely disable the environments feature by defining the following registry key: Path: `HKEY_LOCAL_MACHINE\SOFTWARE\National Instruments\TestStand\`, or 64-Bit `HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\National Instruments\TestStand\` Name: `DisableTestStandEnvironments`

Type: `DWord`

Value: `0x1`

An error will occur if you enable this registry key and launch a TestStand application that attempts to use an environment other than the global environment. This setting applies to all installed versions and bitnesses of TestStand.

- **Remote Engine**—The TestStand remote engine does not support environments. The remote engine always runs in the global environment.
- **Simultaneous Execution**—Although it is possible to run two or more instances of the TestStand engine simultaneously in different environments, the instances are not completely isolated from each other. Certain features of TestStand are shared at runtime, and could conflict. For example:
 - Out-of-process synchronization objects are global to the system and not

- scoped by environments. (In-process synchronization objects are private to the process and therefore cannot conflict across environments.)
- Out-of-process execution for LabVIEW and CVI Adapter steps will generally occur within a single shared instance of the external LabVIEW or CVI ADE. (In-process execution remains private to the process, so it cannot conflict.)
- 64-bit TestStand applications configured to use the 32-bit NI Switch Executive (NISE) share a single instance of the out-of-process NISE. 64-bit TestStand applications configured to run NISE in-process will not conflict. 32-bit TestStand applications always run NISE in-process, and therefore cannot conflict.
- When debugging, stepping into a code module from an adapter step can use an existing instance of the external ADE already in use by a TestStand application running in a different environment.

Changing Environments

Changing environments requires exiting the existing application instance and launching a new one. You can do this manually by exiting the application and launching a new application instance, specifying the new environment on the command line.

For convenience, the TestStand Sequence Editor and full featured User Interfaces support changing environments directly from the Configure Environment dialog.

Select **Configure»Environment** to launch the Configure Environment dialog box. From there, load the desired `.tsenv` file, and choose **Set Engine Environment**. When prompted to end all executions, close all sequence files, and restart the engine, click **OK**. The current application instance shuts down. A new instance launches in the specified environment.

Enabling Large Address Aware Functionality in 32-bit TestStand

Because 32-bit Microsoft Windows applications can use only 2 GB of memory by default, you might encounter memory limits when developing memory-intensive TestStand applications. To increase the amount of memory 32-bit TestStand can use,

you can configure 32-bit TestStand to be large address aware.



Note The Large Address Aware flag is enabled by default for the TestStand Sequence Editor and TestStand User Interfaces, as well as for any executables created with LabVIEW or LabWindows/CVI.

When you enable the Large Address Aware flag for an executable file, you increase the maximum amount of memory available to a 32-bit application from 2 GB of memory to 3 GB of memory on 32-bit Windows, and up to 4 GB of memory on 64-bit Windows.

If you are using 32-bit Windows, you must configure Windows to support large address aware applications to access the additional memory. Refer to the Microsoft Developer Network website for additional information about configuring Windows to be large address aware using the Microsoft 4-Gigabyte Tuning feature.



Note NI recommends using Windows 7 or later with large address aware functionality.

Visit ni.com/info and enter the Info Code `exfu2i` to access the NI support article 5ZREUK2G, Using the Large Address Aware Flag With TestStand Applications, for more information about using the large address aware functionality.

Extending TestStand

You can configure TestStand using the following components:

- **Test Station Options**—You can customize options that apply to all TestStand applications.
- **TestStand Sequence Editor Options and Panes**—You can customize options and the panel and document layout of the sequence editor.
- **TestStand User Interface Options**—You can customize options for user interfaces.
- **Tools Menu**—You can customize the Tools menus in the TestStand Sequence Editor and User Interfaces to contain exactly the tools you need.
- **Directory Structure**—You can customize the search path TestStand uses to locate files specified by relative paths.
- **Adapters**—You can configure unique settings for each adapter, such as specifying which run time to use or whether to display parameter information.
- **Result Processing and Model Options**—

You can also customize the following TestStand components:

- **Data Types**—You can customize copies of the standard data types or create and modify your own data types
- **Step Types**—You can customize copies of the standard step types or create and modify your own step types.
- **Callbacks**—You can modify callbacks to customize the operation of test stations.
- **Process Model Plug-ins**—TestStand includes process model plug-ins that perform functions such as reporting and database logging. You can create custom process model plug-ins to extend the functionality of all process models.
- **User Interfaces**—TestStand includes full source code in several different programming languages so you can modify the user interfaces to meet the specific needs of your application.
- **Process Models**—TestStand includes fully customizable Sequential, Parallel, and Batch process models to meet specific testing needs.
- **Sequence File Translators**—You can create sequence file translators to generate and load a sequence file from a test description file saved in a custom format, such as XML or text.
- **TestStand Sequence Analyzer Rules**—You can customize options for built-in rules

and create custom rules to evaluate during sequence analysis.

Configure Menu

Use the Configure menu in the TestStand Sequence Editor and in a TestStand User Interface to control the operation of the TestStand station.

Related concepts:

- [Customizing the Tools Menu](#)
- [Data Types](#)
- [Type Concepts](#)
- [Custom Step Types](#)
- [Customizing Process Models and Callbacks](#)
- [Process Model Plug-In Architecture](#)
- [Creating Custom User Interfaces](#)

Customizing the Tools Menu

The Tools menu in the TestStand Sequence Editor and User Interfaces contains common tools for use with TestStand. You can modify the Tools menu to contain exactly the tools you need, and you can also use the Customize Tools Menu dialog box to add new items to the Tools menu.

Configuring Sequence Editor and User Interface Startup Options

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

Option	Purpose
<code>sequencefile {sequencefile2}...</code>	Instructs the application to automatically load the sequence files at startup, as shown in the following example: <code>SeqEdit.exe "c:\My Seqs\seq1.seq" "c:\My Seqs\seq2.seq"</code>
<code>/run sequencefile</code>	Instructs the application to automatically load and run the sequence in the sequence file at startup, as shown in the following example: <code>SeqEdit.exe /run MainSequence "c:\My Seqs\test.seq"</code>
<code>/runEntryPoint entrypointnamefile</code>	Instructs the application to automatically load and run the sequence file at startup using the Execution entry point you specify, as shown in the following example: <code>SeqEdit.exe /runEntryPoint "Test UUTs" "c:\My Seqs\test.seq"</code> You must use the entry point name to specify the entry point to use, not the sequence name. The entry point name is displayed in the Execute menu in the sequence editor. To change the entry point name, edit the Entry Point Name Expression field, located in the Model tab of the Sequence options.
<code>/editor</code>	Instructs the application to open in Editor Mode when the application supports editing, as shown in the following example: <code>testexec.exe /editor</code>
<code>/goto location</code>	Instructs the application to display the item the location property object path specifies, as shown in the following example: <code>testexec.exe c:\example.seq /goto "Seq[1].Main[4].TS.LoadOpt"</code>
<code>/operatorInterface</code>	Instructs the application to open in Operator Mode, as shown in the following example: <code>testexec.exe /operatorInterface</code>
<code>/quit</code>	Instructs the application to exit after running the executions you specify, as shown in the following example: <code>SeqEdit.exe /run MainSequence "c:\My Seqs\test.seq" /quit</code> TestStand ignores the <code>/quit</code> option if the execution fails to launch.

Option	Purpose
<code>/useExisting</code>	Instructs the application to use the existing running instance of the application instead of opening a new instance, as shown in the following example: <code>SeqEdit.exe /useExisting</code> TestStand ignores the <code>/useExisting</code> option when you specify the <code>/quit</code> option.
<code>/setCurrentDir</code>	Instructs the application to set the current directory to the first directory in the File dialog box directory history list, as shown in the following example: <code>SeqEdit.exe /setCurrentDir</code> The current directory is the directory the File dialog box initially displays when you open or save a file. Use this option to instruct the File dialog box to display the directory the File dialog box displayed the last time you ran the application. TestStand sets the current directory after processing the other command-line options.
<code>/?</code>	Instructs the application to launch a help dialog box that contains a list of valid command-line arguments and then close immediately, as shown in the following example: <code>SeqEdit.exe /?</code> TestStand ignores all other options when you specify the <code>" / ? "</code> option.
<code>/outputToStdIO</code>	Instructs the application to write the status of completed executions as standard output, as shown in the following example: <code>SeqEdit.exe /outputToStdIO</code> When you run a user interface from the command line, the console window displays the standard output from the application.
<code>/env <environment path></code>	Launches the application in the environment defined in the <code>.tsenv</code> file located at <code><environment path></code> .
<code>workspacefile</code>	Instructs the application to automatically load the workspace file at startup, as shown in the following example: <code>SeqEdit.exe "c:\My Workspaces\ws1.tsw"</code> This option applies only to the Sequence Editor.
<code>sequenceanalyzerprojectfile</code>	Instructs the application to automatically load the sequence analyzer project file at startup, as shown in the following example: <code>SeqEdit.exe "c:\My spaces\myAnalyzerFile.tsaproj"</code> This option applies only

Option	Purpose
	to the Sequence Editor.

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

Related concepts:

- [Creating Editor Applications](#)
- [Editor Versus Operator Interface Applications](#)
- [Application Manager](#)

TestStand Directory Structure

To comply with Microsoft Windows restrictions on writing to the `Program Files` directory, TestStand 4.1 or later installs some files in different locations from previous versions of TestStand. Refer to the Directory Relocation section of the NI TestStand Version 4.2 Release Notes for more information about specific directories and files relocated in TestStand 4.1 or later.

TestStand 2014 or later installs files in the following default directories:

Directory	32-bit TestStand	64-bit TestStand
<TestStand>	(32-bit Windows) <code>C:\Program Files\National Instruments\TestStand <version></code> (64-bit Windows) <code>C:\Program Files (x86)\National Instruments\TestStand <version></code>	<code>C:\Program Files\National Instruments\TestStand <version></code>
<TestStand Public>	(Windows 10/8.1/7) <code>C:\Users\Public\Documents\National Instruments\TestStand <version></code> (32-bit)	<code>C:\Users\Public\Documents\National Instruments\TestStand <version></code> (64-bit)

Directory	32-bit TestStand	64-bit TestStand
<TestStand Application Data> (Hidden by default)	(Windows 10/8.1/7) C:\ProgramData\National Instruments\TestStand <version> (32-bit)	C:\ProgramData\National Instruments\TestStand <version> (64-bit)
<TestStand Local Application Data> (Hidden by default)	(Windows 10/8.1/7) <User Directory>\AppData\Local\National Instruments\TestStand <version> (32-bit)	<User Directory>\AppData\Local\National Instruments\TestStand <version> (64-bit)

TestStand adds the TestStand, TestStandAppData, and TestStandPublic environment variables that reference the <TestStand>, <TestStand Public>, and <TestStand Application Data> directories respectively. You can use the environment variables in development environments to specify paths to files in these TestStand directories. For example, you can use these environment variables to build paths in the **Additional Include Directories** control in Visual Studio.

Refer to the Defining an Environment topic for more information about using environment variables in side-by-side TestStand environments.

Related concepts:

- [TestStand Directory](#)
- [TestStand Public Directory](#)
- [<TestStand Application Data> Directory](#)
- [TestStand Local Application Data Directory](#)
- [Defining an Environment](#)
- [TestStand Environments](#)

TestStand Directory

The following table shows the name and contents of each subdirectory of the <TestStand> directory, which is where you installed TestStand on the computer and contains the read-only program files.

Table 38. Contents of the TestStand Directory

Directory Name	Contents
AdapterSupport	Read-only support files for the .NET and HTBasic Adapters.
API	Read-only TestStand ActiveX Automation server libraries and utility libraries for several programming languages.
Bin	Read-only TestStand Sequence Editor executable, TestStand Engine DLLs, and support files.
CodeTemplates	Read-only source code templates for step types.
Components	Read-only components installed with TestStand, including callback files, converters, icons, language files, process model files, step types, source files, compatibility files, utility files, and sequence analyzer files.
Doc	Documentation files.
UserInterfaces	Read-only LabVIEW, LabWindows/CVI, Microsoft Visual Basic, C#, and C++ (MFC) user interfaces with source code.

Related concepts:

- [Components Directory](#)

Components Directory

TestStand installs the default sequences, executables, project files, and source files for components in the following subdirectories of the <TestStand>\Components directory.

Directory Name	Contents
Analyzer	Contains TestStand Sequence Analyzer built-in rules file, built-in analysis modules, and analysis module templates. You can customize sequence analyzer rules.
Callbacks	Contains the sequence files in which TestStand stores Station Engine callbacks and Front-End callbacks.
Compatibility	Contains type palette files TestStand uses to save sequence files compatible with earlier versions of TestStand.
Icons	Contains icon files for module adapters and step types.
Language	Contains string resource files in language-specific subdirectories.

Directory Name	Contents
Models	Contains the default process model sequence files that you can customize, process model plug-ins, and supporting code modules.
Obsolete	Contains components TestStand no longer uses but installs to maintain compatibility.
Schemas	Contains the schema for XML-formatted TestStand sequence files.
StepTypes	Contains support files for built-in step types. TestStand installs support files for the built-in step types in the <TestStand>\Components\StepTypes directory. You can also customize step types.
Stylesheets	Contains the style sheet and supporting files to view sequence analyzer reports and TestStand File Diff and Merge utility reports.
Tools	Contains sequences and supporting files for the Tools menu commands.
Translators	Contains sequence file translator DLLs. TestStand does not create this directory by default.
TypePalettes	Contains the default type palette files.

The TestStand Engine searches for sequences and code modules using the TestStand search directory path. The default search precedence places the <TestStand Public>\Components directory before the <TestStand>\Components directory to ensure that TestStand loads the sequences and code modules you customize instead of loading the default TestStand versions of the files. Select **Configure»Search Directories** from the sequence editor menu bar to modify the precedence of the search directory paths.

When you use the TestStand Deployment Utility to deploy a run-time version of the TestStand Engine, you can bundle customized components in the <TestStand Public> directory with the TestStand run-time deployment.

Related concepts:

- [Modifying Engine Callbacks](#)
- [Modifying Front-End Callbacks](#)
- [Module Adapters](#)
- [Creating String Resource Files](#)
- [Process Model Support Files](#)

- [Modifying Process Model Sequence Files](#)
- [Process Model Plug-In Architecture](#)
- [Built-In Step Types](#)
- [TestStand Directory](#)
- [Custom Step Types](#)
- [Customizing the Tools Menu](#)
- [Sequence File Translators](#)
- [Type Concepts](#)
- [Search Directories](#)
- [TestStand Public Directory](#)
- [Deploying TestStand Systems](#)

TestStand Public Directory

The following table lists the name and contents of each subdirectory of the `<TestStand Public>` directory, which contains modifications, customizations, and other files you can directly edit.

Directory Name	Contents
AdapterSupport	Support files for the LabVIEW and LabWindows/CVI Adapters.
CodeTemplates	Empty by default. You can store modified source code templates for step types, or code templates you create.
Components	Placeholder subdirectories for modified TestStand components and components you develop, including callback files, language files, process model files, run-time servers, type palette files, and TestStand Sequence Analyzer analysis modules for custom rules you create. TestStand installs the default sequences, executables, project files, and source files for components in the <code><TestStand>\Components</code> directory.
Examples	Example sequences and tests. Most example sequences that use VIs call subVIs in the <code><LabVIEW>\vi.lib</code> directory, which you can access after you install LabVIEW.
RuntimeServers	Contains example legacy source code for building a LabVIEW run-time application for executing LabVIEW-based code modules on a computer on which you have not installed the LabVIEW development system. Use the LabVIEW Adapter Configuration dialog box to configure the LabVIEW Adapter to use LabVIEW executables built with an ActiveX server enabled.

Directory Name	Contents
Setup	Support files for the TestStand installer.
Tutorial	Sequences and code modules you use in tutorials the Getting Started with TestStand manual and the NI TestStand Evaluation Guide. This directory also contains a <code>Solution</code> subdirectory that contains completed versions of the tutorial files.
UserInterfaces	Copies of the LabVIEW, LabWindows/CVI, Microsoft Visual Basic, C#, and C++ (MFC) user interfaces with source code installed in the <code><TestStand>\UserInterfaces</code> directory.

Related concepts:

- [Programming with the TestStand API in LabVIEW](#)
- [Programming with the TestStand API in LabWindows/CVI](#)
- [Step Types](#)
- [Components Directory](#)
- [Getting Started with TestStand](#)
- [TestStand Directory](#)

Copying Read-Only Files to Modify

By default, TestStand installs source files in the protected and non-writeable `<TestStand>\CodeTemplates`, `<TestStand>\Components`, and `<TestStand>\UserInterfaces` directories.

To modify the installed code templates or components or to create new code templates or components, copy the files from the `<TestStand>` directories to the `<TestStand Public>` directories and make changes to the copies of the files.

To modify the installed user interfaces or to create new user interfaces, modify the files TestStand installs in the `<TestStand Public>\UserInterfaces` directory.

Guidelines for Modifying Copied Files

Use the following guidelines when you copy installed files to modify:

- You must rename the files after you modify them if you want to create a separate

- custom component. You must register any new or renamed ActiveX components.
- You do not have to rename the files after you modify them if you only want to modify the behavior of an existing component.
- If you do not rename the files and you use the files in a future version of TestStand, changes NI makes to the component might not be compatible with the modified version of the component.
- Storing new and customized files in the <TestStand Public> directory ensures that new installations of the same version of TestStand do not overwrite the customizations and ensures that uninstalling TestStand does not remove the files you customize.



Note If you modify any process model or process model plug-in types, open and save all the process models and process model plug-ins that use the modified types to ensure you avoid a type conflict. To modify or save the process models, you must first copy all the files from the <TestStand>\Components\Models\TestStandModels directory to the <TestStand Public>\Components\Models\TestStandModels directory and modify or save the files in the <TestStand Public>\Components\Models\TestStandModels directory. The files in the <TestStand Public>\Components\Models\TestStandModels directory override the files in the <TestStand>\Components\Models\TestStandModels directory. Similarly, you must copy any process model plug-in files from the <TestStand>\Components\Models\ModelPlugins directory to the <TestStand Public>\Components\Models\ModelPlugins directory before modifying the files.

Related concepts:

- [TestStand Directory Structure](#)

<TestStand Application Data> Directory

The <TestStand Application Data> directory is hidden by default and contains configuration files users generally do not edit but an application can edit. The directory includes the Cfg subdirectory, which contains configuration files for TestStand Engine, TestStand Sequence Editor, and TestStand User Interface options, as

shown in the following table. You might need to deploy these files to a target computer when creating a deployment.

Table 39. <TestStand Application Data> Directory Contents

Directory Name	Contents
SeqEdit directory	Default and saved layout files for the sequence editor.
Analyzer.ini	Settings for the TestStand Sequence Analyzer.
ConnectionSource.ini	Datalink list the Database and Legacy Property Loader step types use for configuration.
CustomRules.tsarules	Information about custom sequence analyzer rules and analysis modules.
DatabaseViewerOptions.ini	Configuration options for the Database Viewer application.
Deployment Utility RecentFiles.ini	Most recently used file list for the TestStand Deployment Utility.
DeploymentUtilityOptions.ini	Configuration options for the deployment utility.
DocGen.ini	Configuration options for the Sequence File Documentation tool.
EngineParts.ini	Deployment utility uses this file when you include the TestStand Engine in an installer.  Note Do not edit this file.
ExpressMenu	Cache of Express VI menu for the specified LabVIEW version.
Filter.ini	List of files to exclude from deployment.
ModelPlugins\ ResultProcessing.cfg	Configuration options for result processing model plug-ins.
PLPropertyLoader.ini	Configuration options for the Legacy Property Loader Import/Export Properties tool.
ImportExportTool.ini	Configuration options for the Import/Export Properties tool.
StationGlobals.ini	Definitions and values of station global variables.

Directory Name	Contents
<code>Templates.ini</code>	Templates configuration for the Insertion Palette in the sequence editor.
<code>Adapters.cfg</code>	Settings related to the TestStand adapters.
<code>SearchDirectories.cfg</code>	Specifies TestStand search directories.
<code>TypePalettes.cfg</code>	Specifies locations of type palette files.
<code>StartupCfg.ini</code>	Settings needed prior to engine construction.
<code>GeneralEngine.cfg</code>	General engine settings including most station options.
<code>TestStandModelOptions.ini</code>	Model options for built-in process models.
<code>TestStandDatabaseSchemas.ini</code>	Database schemas for result logging.
<code>TestStandPersistedOptions.opt</code>	Persisted breakpoints and watch expressions to use when no workspace is loaded.
<code>ToolMenu.ini</code>	List of Tools menu items.
<code>Users.ini</code>	List of TestStand users.
<code>XmlPack.ini</code>	Contains a comma-separated string of file extensions for files you want to distribute using the XML Packaging Utility.



Note The TestStand 2012 or later process models include significant changes to implement a process model plug-in architecture for model sequence files. TestStand 2012 or later uses the plug-in mechanism to decouple result processing from the process models. As part of this change, the result processing options move from `TestStandModelReportOptions.ini` and `TestStandDatabaseOptions.ini` into a single `ResultProcessing.cfg` file, located in the `<TestStand Application Data>\Cfg\ModelPlugins` directory, that stores configuration information for all result processing model plug-ins. The database schemas that were stored in `TestStandDatabaseOptions.ini` are stored in `TestStandDatabaseSchemas.ini`, separate from `ResultProcessing.cfg`. The result processing options files from

TestStand 2010 SP1 or earlier are not compatible with TestStand 2012 or later but can be used with the equivalent legacy TestStand 2010 process models. Use the TestStand Migration Utility to migrate the result processing options from TestStand 2010 SP1 or earlier installed on the computer to the current version of TestStand.

Related concepts:

- [Creating Deployments](#)
- [Deploying TestStand Systems](#)
- [Customizing the Tools Menu](#)
- [Filtering and Excluding Files](#)

TestStand Local Application Data Directory

The <TestStand Local Application Data> directory is hidden by default and contains configuration files users generally do not edit but an application can edit, as shown in the following table.

Table 40. Contents of the TestStand Local Application Data Directory

Directory Name	Contents
Layout_current.bin directory	Layout of the TestStand Sequence Editor the last time it was closed, including window position, size, docked state, and other settings for the panes, menus, and toolbars.
SeqEdit.xml	Configuration options for the sequence editor, such as panel size and font settings used on the Insertion Palette pane and the Variables pane.
UserInterface.xml	Configuration options for TestStand User Interfaces, such as window and panel sizes, font settings, and sequences to load. You can customize the data stored in this file.

Type Concepts

TestStand stores step type and data type definitions in files and in memory. You can modify types and use type version numbers to determine which version of the type to load. Use the Types window to create, modify, and examine step types and data types. You can password-protect types so that other TestStand users cannot modify the types

in the TestStand Sequence Editor.

Related concepts:

- [Step Types](#)
- [Standard and Custom Data Types](#)
- [Modifying Type Instances, Values, and Type Definitions](#)
- [Type Versioning](#)
- [Password-Protecting Types](#)

Storing Types in Files and Memory

TestStand files store the definition for each step type and data type the file uses. You can also specify that a file always saves the definition for a type, even when the file does not currently use the type. Because many files can use the same type, many files can contain the definition for the type. All sequence files, for example, might contain the definitions for the Pass/Fail Test step type and the Error standard data type.

TestStand allows only one definition for each uniquely named type in memory. The type can appear in multiple files, but only one underlying definition of the type exists in memory. When you modify the type in one file, the type definition updates in all loaded files.

Related concepts:

- [Error and CommonResults](#)
- [Modifying Type Instances, Values, and Type Definitions](#)

Password-Protecting Types

You can password-protect types so that other TestStand users cannot modify the types in the TestStand Sequence Editor.

In the Types window, select one or more types you want to password-protect, right-click, and select **Password Protect** from the context menu to launch the Password Protect Type Definitions dialog box, in which you can enable or disable password protection and change the password settings for the selected types. Enable the

Password Protection option to enable password protection for the types, enter a password in the **New Password** control, and verify the password in the **Re-enter Password** control.

After you enable password protection for the types, the types remain unlocked so you can continue to edit the types. TestStand automatically locks password-protected types when TestStand first loads the types. For example, when you restart TestStand or close and reopen all the files that reference the password-protected types, TestStand locks the types. When you restart TestStand, TestStand locks all password-protected types. You can also select one or more password-protected types, right-click, and select **Lock** from the context menu to explicitly lock the types.

To unlock types, select the types you want to unlock, right-click, and select **Unlock** from the context menu to launch the Unlock Type Definitions dialog box, in which you can enter the correct password to unlock any selected types to which the password applies. TestStand ignores any selected types that are already unlocked or for which you have not enabled password protection. If the password you enter does not apply to all of the selected types, TestStand launches the Unlock Failed for Some Type Definitions dialog box, which gives you the option to return to the Unlock Type Definitions dialog box and enter another password to unlock the remaining types.

TestStand also launches the Unlock Type Definitions dialog box when you attempt to edit a locked type. You must enter the correct password to unlock the type before you can edit the type.

The locked or unlocked state of a type is not a saved state but is instead an in-memory state that indicates whether TestStand currently permits editing of the password-protected type.



Note

- TestStand supports password-protecting types to deter unauthorized users from editing the types in the sequence editor. However, any TestStand user can continue to programmatically edit a locked type by using the TestStand PropertyObject API. NI does not recommend password-protecting types as the only way of protecting intellectual

property.

- If you look at least one type in an INI-formatted sequence file, the entire contents of the sequence file become protected and unreadable.

Modifying Types

You can modify the built-in and custom properties of step types you create and custom data types you create. However, you cannot modify the built-in step types and standard data types TestStand installs.

Use the Copy and Paste context menu items to copy and rename an existing or built-in step type or standard data type on the Types pane of the Types window of the TestStand Sequence Editor.

When you modify a type, TestStand enables the `PropTypeFlags_IsModifiedType` type flag for the type. TestStand cannot automatically resolve type conflicts unless you disable the `PropTypeFlags_IsModifiedType` type flag. To disable the `PropTypeFlags_IsModifiedType` type flag, you typically increment the version number of the type on the Version tab of the Step Type Properties dialog box or on the Version tab of the Type Properties dialog box when you complete all the modifications to the type.

By default, the Before Saving Modified Types option on the File tab of the Station Options dialog box is set to Prompt to Increment Type Versions. This causes TestStand to launch the Modified Types Warning dialog box when you select **File»Save <filename>** and the sequence file or type palette contains types that are marked as modified. You can increment the type version or remove the modified mark on the type before saving, or save the type as modified.

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



Note If you modify any process model or process model plug-in types, open and save all the process models and process model plug-ins that use the modified types to ensure you avoid a type conflict. To modify or save the process models, you must first copy all the files from the `<TestStand>\Components\Models\TestStandModels` directory to the `<TestStand Public>\Components\Models\TestStandModels` directory and modify or save the files in the `<TestStand Public>\Components\Models\TestStandModels` directory. The files in the `<TestStand Public>\Components\Models\TestStandModels` directory override the files in the `<TestStand>\Components\Models\TestStandModels` directory. Similarly, you must copy any process model plug-in files from the `<TestStand>\Components\Models\ModelPlugins` directory to the `<TestStand Public>\Components\Models\ModelPlugins` directory before modifying the files.

Related concepts:

- [Creating Custom Step Types](#)
- [Creating Custom Data Types](#)
- [Built-In Step Types](#)
- [Standard and Custom Data Types](#)
- [Managing Type Revisions](#)
- [Type Versioning](#)
- [TestStand Directory Structure](#)

Modifying Types Without Altering the Version Number

You can use `PropertyObject` Attributes to add data to or update data in a custom data type without needing to increment the version number of the custom type.

Since object Attributes are an unstructured container, creating an Attribute within an instance of a custom type does not generate a type conflict. This allows you to use the type in other TestStand versions which run on the same engine version, making the types more version-agnostic. This also allows you to use sequence files that call custom types in multiple versions of TestStand without needing to save the sequence file for each version using **Save As**, as long as the TestStand versions run on the same

engine version.



Note You can check the engine version of a version of TestStand by consulting the program information at **Help » About TestStand**.

If you need to use a custom type Attribute in a test sequence, you can create the Attribute in the unique instance of the custom type. If you use this method, you should write your test programs to ensure they function if the Attribute isn't created for a type instance.

If the runtime behavior of a step which calls your custom type changes when run on older versions of TestStand with the same engine version, you should add an attribute to the type which uses the TestStand Sequence Analyzer to report the behavior change or error. Add the following attributes to the type instance to report an error using an Analyzer message and the `NI_NotSupportedFeature` built-in Analyzer rule:

Figure 1. Type Instance Unsupported Feature Analyzer Message Structure

NAME	VALUE	TYPE	COMMENT
Attributes			
TestStand		Container	▼
UnsupportedFeatures		Array of Containers[0..0]	▼
[0]		Container	▼
AnalyzerMessage	"Updated Analyzer msg"	String	▼
FeatureId	"SweepLoopUpdate"	String	▼
MaximumTestStandVersion	"24.0"	String	▼
MinimumTestStandVersion	"22.0.0.49152"	String	▼
<Right click to insert Attribute>			
<Right click to insert Attribute>			

These attributes should contain the following information:

- **UnsupportedFeatures:** Should be structured as an array where each element represents a feature and contains the following:
 - **FeatureId (String):** Unique identifier for the feature.
 - **AnalyzerMessage (String):** Message you want to pass to users when the feature is unsupported.
 - **MaximumTestStandVersion (String):** Maximum TestStand version that the feature can be used in. This value should follow the `major.minor.revision.build` format. Revision and build version values are optional.
 - **MinimumTestStandVersion (String):** Minimum TestStand version that the feature can be used in. This value should follow the `major.minor.revision.build` format. Revision and build version

values are optional.

If a user tries to use a Type feature outside of the range of supported TestStand versions, the Sequence Analyzer will report an error message with the following structure:

```
Message: The instance of type <TypeName> has some features which are not supported  
in current TestStand version: <AnalyzerMessage>.
```

Related concepts:

- [Modifying Types](#)
- [TestStand Sequence Analyzer](#)

Type Versioning

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

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

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

Use the **Set Earliest TestStand Versions that can Use this Type** option on the Version tab of the Type Properties dialog box to specify the earliest TestStand version that can use a type to prevent the TestStand Engine from using the type when the version of the engine is earlier than the TestStand version you specify. When you enable this option

and an earlier version of the engine attempts to load the type, TestStand ignores the type and loads the file only when an earlier version of the type already exists in memory. If you do not specify the earliest TestStand version that can use a type and you set the Allow Automatic Type Conflict Resolution option on the File tab of the Station Options dialog box to Always, future versions of a type might propagate into type palette files and sequence files that run in the previous TestStand version.

Related concepts:

- [Avoiding Unwanted Type Version Propagation](#)
- [Managing Type Revisions](#)

Managing Type Revisions

When you load a file that contains a type definition and another type definition with the same name already exists in memory, TestStand compares the two type definitions, including all the built-in and custom subproperties in the types.

If the types are identical, TestStand continues to use the type in memory. If the types are not identical, TestStand attempts to resolve the type conflict. TestStand automatically selects the type with the latest version number when all of the following conditions exist:

- The version numbers of the types are different
- The `PropTypeFlags_IsModifiedType` type flag is disabled for both types
- The **Always prompt the user to resolve the conflict** option on the Version tab of the Step Type Properties dialog box or on the Version tab of the Type Properties dialog box is disabled for both types
- The **Allow Automatic Type Conflict Resolution** option on the File tab of the Station Options dialog box does not restrict automatic type conflict resolution in this situation

If TestStand cannot automatically resolve a type conflict, TestStand either launches the Type Conflict in File dialog box or, if TestStand is loading a file an execution calls, returns a run-time error.



Note If an execution calls a file that has a later version of a type than the

type version currently loaded in memory, TestStand does not automatically resolve the conflict and instead returns a run-time error. Replacing a type definition while an execution is running is unsafe because the execution might be using the type.

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



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

Avoiding Unwanted Type Version Propagation

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

- You create a new version of a step type that is not compatible with previous versions of TestStand. A sequence file that contains the new type loads in a previous version of TestStand that contains an older version of the type in a type palette. In versions of TestStand earlier than version 4.1, the new version of the type automatically propagates to the type palette in the previous version of TestStand and to all files loaded in memory from then on. Because the type is not compatible with previous versions of TestStand, you might experience undesired behavior when you run sequences that use the new type. In TestStand 4.1 or later, by default types no longer automatically propagate to type palette files.
- Two types with completely different functionality share the same name. When the TestStand Engine loads the two types, the engine assumes that the types are different versions of the same type and loads the type with the later version number, which can affect the functionality of the sequence files that contain instances of the second type.

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

Avoiding Unwanted Type Version Propagation to Previous Versions of TestStand

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

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

Related concepts:

- [Managing Type Revisions](#)
- [TestStand Directory Structure](#)

Using Type Palette Files to Restrict Automatic Type Conflict Resolution

To avoid unwanted type version propagation, create all types in type palette files and use the default behavior of the Allow Automatic Type Conflict Resolution option on the File tab of the Station Options dialog box.

Common Scenarios for Allow Automatic Type Conflict Resolution Options

The following table lists each of the Allow Automatic Type Conflict Resolution options on the File tab of the Station Options dialog box and common scenarios for using each option.

Allow Automatic Type Conflict Resolution Option	Common Scenarios for Option
Only if Type Palette Files will not be Modified (default)	- You want type palette files to be the only location in which you strictly define types. - You do not want to allow any changes to types located in type palette files without the user explicitly deciding to do so.
Only if a Type Palette File has the Higher Version	- You want type palette files to be the only location in which you strictly define types. - You do not want to allow any changes to types located in type palette files without the user explicitly deciding to do so. - You do not want to allow automatic conflict resolution for types that exist only in sequence files and not in type palette files to avoid mismatched versions of types.
Always (same as TestStand 4.0.x or earlier)	None. Use this option only for compatibility with versions of TestStand earlier than version 4.1.
Never	You want to ensure that all files use exactly the same version of all their types.

Type Palette Files

Type palette files contain step types, standard data types, and custom data types you want available in the TestStand Sequence Editor or in the TestStand User Interfaces at all times. Drag a type to a type palette file in the Types window to ensure that the type is always available, even when the User Manager window, station global variables, or any open sequence files do not use the type.

Type palette files are located in the `<TestStand>\Components\TypePalettes` directory. Typically, you create new types in the `MyTypes.ini` type palette file in the `<TestStand Public>\Components\TypePalettes` directory or in a new type palette file you create.

You can distribute step types and data types you create to other computers by copying a type palette file to the `<TestStand Public>\Components\TypePalettes` directory. Right-click the **View Types For** pane in the Types window and select **Customize Type Palettes** from the context menu to launch the Configure Type Palettes dialog box, in which you can specify which type palette files TestStand uses.

Adding Type Palette Files

Use one of the following methods to add a type palette file to an existing installation.

- Copy the file to the `<TestStand Public>\Components\TypePalettes` directory. When you launch the sequence editor or a user interface, TestStand searches this directory for new files and adds those files to the type palette list.



Note TestStand will not add a type palette file that you explicitly excluded from the type palette list, even if you add the file to the `<TestStand Public>\Components\TypePalettes` directory.

- Copy the file to the `<TestStand Public>\Components\TypePalettes` directory and add the prefix `Install_` to the file name. When you launch the sequence editor or a user interface, TestStand identifies files with the `Install_` prefix and takes one of the following actions:
 - If TestStand locates a type palette file in the type palette list with the same name, but without the `Install_` prefix, TestStand merges the types in the `Install_` prefix version of the file with the types in the existing file and then deletes the `Install_` prefix version.
 - If TestStand locates a type palette file with the same name, but without the `Install_` prefix, that is not included in the type palette list because a user explicitly excluded it, TestStand does not change the contents of either file.



Note To ensure that TestStand always installs a type palette file, even when previously excluded from the type palette list, you must

first delete the original version of the file—which does not contain the `Install_` prefix—before you copy the `Install_` prefix version to the `<TestStand Public>\Components\TypePalettes` directory.

- If TestStand does not locate a type palette file with the same name, but without the `Install_` prefix, TestStand creates a new file with the same name and adds it to the type palettes list. TestStand then adds the types in the `Install_` prefix version of the file to the new type palette file and deletes the `Install_` prefix version.
- Click **Add** in the Configure Type Palettes dialog box to manually add the type palette file.

Related concepts:

- [Station Global Variables](#)
- [TestStand Directory Structure](#)
- [Deploying Type Palette Files](#)

Data Types

When you insert variables, parameters, or step properties, you can select a data type to use for the item. You can also create and modify custom data types to meet the needs of an application.

Related concepts:

- [Modifying Type Instances, Values, and Type Definitions](#)
- [Creating Custom Data Types](#)
- [Type Concepts](#)

Creating Instances of Data Types

The context menu of each window or pane in which you can insert a variable, parameter, or property includes an Insert item, as listed in the following table.

Context Menu Item	Local of Context Menu	Item Inserted
Insert File Global	File Globals section of the Variables pane in the Sequence File window	Sequence file global variable
Insert Parameter	Parameters section of the Variables pane in the Sequence File window	Sequence parameter
Insert Local	Locals section of the Variables pane in the Sequence File window	Sequence local variable
Insert Station Global	Station Globals window	Station global variable
Insert UserInsert Group	User Manager window	New object with the User data type
Insert Field	Types window	New element in an existing data type

With the exception of the Insert User and Insert Group items, all the context menu items in the preceding table provide a submenu from which you can select the following categories of data types:

- Basic types, such as number, string, Boolean, and object reference.
- The Container type, in which you can insert subproperties of any type.



Note You can use an empty container as a parameter when you want to pass an object of any type to the sequence, in which case you also must turn off type checking for the parameter.

- Named data types, including all the custom named data types in type palette files or in the file you are currently editing. The submenu also includes standard TestStand named data types, such as Error, CommonResults, Path, Expression, and NI_TDMSReference.
- An array of elements of one of the types described above.

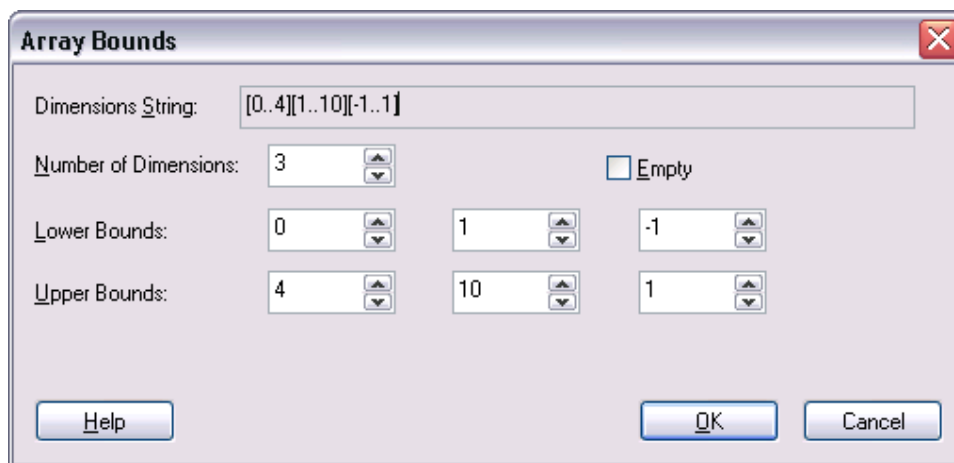
You can also create a new, custom data type in the Types window and then select the new data type from the Type submenu of the Insert context menu.

Related concepts:

- [Using Standard Named Data Types](#)
- [NI_TDMSReference](#)
- [Creating and Deleting Users Using API](#)

Specifying Array Sizes

In the context menu of each window or pane in which you can insert a variable, parameter, or property, select **Insert<item>»Array of»<data type>** to insert an array. When you insert an array, the TestStand Sequence Editor launches the Array Bounds dialog box, in which you can set and modify the array bounds. The following figure shows the settings for a three-dimensional array.



The first and outermost dimension contains five elements, with 0 as the minimum index and 4 as the maximum index. The second dimension contains 10 elements, with 1 as the minimum index and 10 as the maximum index. The third and innermost dimension contains three elements, with -1 as the minimum index and 1 as the maximum index.

After you create the variable, parameter, or property as an array, you can modify the array bounds by clicking the **Resize Array** button in the Name column of the list view to launch the Array Bounds dialog box.

Dynamic Array Sizing

You can resize an array during execution. In an expression, use the `GetNumElements` and `SetNumElements` expression functions to obtain and modify the upper and lower bounds for a one-dimensional array. For multi-dimensional arrays or to change the

number of dimensions in the array, use the `GetArrayBounds` and `SetArrayBounds` expression functions. The Operators/Functions tab of the Expression Browser dialog box includes documentation for each expression function.

In a code module, use the `PropertyObjectType.ArrayDimensions` method to obtain or specify the upper and lower bounds of an array or to change the number of dimensions in the array.

Specifying an Empty Array

Defining an initially empty array is useful when you do not know the maximum array size the sequence requires during execution and when you want to save memory during the periods of execution when the sequence does not use the array. Enable the **Empty** option in the Array Bounds dialog box when you want the array to be initially empty. When you enable this option, the Upper Bounds control for each dimension dims.

You must use the TestStand API to both resize and insert elements into empty arrays.

Related concepts:

- [Dynamic Array Sizing](#)

Modifying Type Instances, Values, and Type Definitions

You can modify the default type instances, values, and type definitions of TestStand data types.

Type Instances

With the exception of resizing arrays, you cannot change the subproperty structure of a variable, parameter, or property after you create it from a data type.

You can, however, change the subproperty structure of the data type itself. Changing the structure of a data type affects all variables, parameters, and properties that use the data type.

Values

Use the Value column on the Variables pane to modify the value of a variable, parameter, or property. For local variables, the data type value you specify at edit time is the initial value TestStand uses whenever it calls into the sequence. For parameters, the data type value is the default value TestStand uses when you do not pass an argument value explicitly.

Type Definitions

On the Types pane of the Types window, right-click a property on a type and select **Properties** from the context menu to launch the Type Properties dialog box, in which you can specify additional settings and edit the value of the property on the type.

In general, when you make changes to property values in a type, the changes do not affect all instances of the type. Enable the **Apply Changes in this Dialog to all Loaded Instances of the Type** option in the Type Properties dialog box to apply the change to all loaded instances of the type.

You can also rearrange properties on the Variables pane by dragging or copying the items you want to move. In general, the order of properties does not matter, but the order of sequence parameters affects how you configure a Sequence Call step that invokes the sequence.



Note If you modify any process model or process model plug-in types, open and save all the process models and process model plug-ins that use the modified types to ensure you avoid a type conflict. To modify or save the process models, you must first copy all the files from the `<TestStand>\Components\Models\TestStandModels` directory to the `<TestStand Public>\Components\Models\TestStandModels` directory and modify or save the files in the `<TestStand Public>\Components\Models\TestStandModels` directory. The files in the `<TestStand Public>\Components\Models\TestStandModels` directory override the files in the `<TestStand>\Components\Models\TestStandModels` directory. Similarly, you must copy any process model plug-in files from the `<TestStand>\Components\Models\ModelPlugins` directory to the

```
<TestStand Public>\Components\Models\ModelPlugins
```

directory before modifying the files.

Related concepts:

- [Specifying Array Sizes](#)
- [TestStand Directory Structure](#)

Assigning Values to Object References

Object reference properties can contain references to LabVIEW, .NET, ActiveX/COM, and other objects. When a property is an object reference, you can use the **Release Object** button, which displays only when the value of the property is non-zero, in the Value column on the Variables pane to release the reference.

You can set the reference value by using an expression, using the TestStand API, or configuring a call to a code module with a parameter or return value that returns such an object.

When you have two object reference properties and use the equality operator in an expression, TestStand performs an equality comparison on the IUnknown pointers for ActiveX objects and the pointer values for .NET objects.

Releasing References to Objects

When you assign an object reference value to an object reference property, TestStand maintains a reference to the object for as long as the property exists, or until the value changes. TestStand automatically releases the reference to the object when the property loses scope. For example, when a sequence local variable contains a reference to an object, TestStand releases the reference when the call to the sequence completes.

You can also release the reference to the object by assigning the property a new value or the constant `Nothing`. Assign the constant `Nothing`, rather than the value `0`, to clear the reference. When you assign `0` to an object reference variable, TestStand creates a numeric variable with a value of `0` and assigns a reference to that numeric variable to the object reference variable.

When you release all references to an object, the availability of the object after release changes depending on the type of object, as shown in the following table.

Type of Object	Object Availability
.NET	The object is available for .NET garbage collection.
ActiveX/COM	The object is destroyed.

Sharing .NET Objects Between LabVIEW and TestStand

You can share .NET objects that derive from `MarshalByRefObject` or that are serializable between LabVIEW and TestStand. You can pass .NET references created in LabVIEW to TestStand and store the references in Object Reference variables in TestStand. You can then use those references with the TestStand .NET Adapter. Similarly, LabVIEW can directly use references you create with the TestStand .NET Adapter. Refer to Microsoft MSDN documentation for more information about `MarshalByRefObject` and serializable objects.



Note This functionality is not supported in TestStand 2024 Q4.

Using Standard Named Data Types

TestStand defines a set of standard named data types, such as `Error`, `CommonResults`, `Path`, `Expression`, and `NI_TDMSReference`. The only standard named data types you can modify are the `CommonResults` and the `NI_UserCustomPrivileges` types.

Related concepts:

- [Error and CommonResults](#)
- [Path](#)
- [Expression](#)
- [NI_TDMSReference](#)

Error and CommonResults

TestStand inserts a `Results` property in every step you add to a sequence. The `Results` property includes at least three subproperties—`Error`, `Status`, and

Common.

Steps use the `Error` subproperty to indicate run-time errors. The `Error` subproperty uses the `Error` standard data type, which is a container that includes `Code`, `Msg`, and `Occurred` subproperties. When a run-time error occurs in a step, the step sets the `Code` subproperty to a value that indicates the source of the error, the `Msg` subproperty to a string that describes the error, and the `Occurred` subproperty to `True`.

The `Common` subproperty uses the `CommonResults` standard data type, which is an initially empty object. By adding subproperties to the `CommonResults` data type, you can add extra result information to all steps in a standard way.

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



Note TestStand 4.1 or later includes code to handle an issue in which sequence files with an unsupported version of `CommonResults` (3.5.0.1) were made publicly available. If you load a file with an empty `CommonResults` 3.5.0.1 data type, TestStand does not automatically use that version of the data type and instead uses the existing global version. This behavior ensures that TestStand does not automatically propagate this unintentionally released version of the type to other files. In addition, when TestStand attempts to load two versions of the `CommonResults` data type and one version of the data type is defined as an empty container and has a version of 3.1.0.100 or earlier, and the other version of the data type is defined as a non-empty container, TestStand does not allow the empty version to automatically override the non-empty version, even if the empty version has a later version number. This behavior ensures that TestStand does not automatically lose changes made to `CommonResults` in versions of TestStand earlier than TestStand 3.1.

When you modify `CommonResults` without incrementing the version number, you might see a type conflict when you open other sequence files, such as `FrontEndCallbacks.seq` when TestStand loads the LoginLogout Front-End

callback sequence before you log in or out. TestStand prompts you to increment the version number when you save changes to any data type or step type. You can avoid the type conflicts by re-saving the type palette files and instructing TestStand to increment the type version for CommonResults before opening any other files. NI recommends modifying the CommonResults data type only when you want to make an architectural change to all step types that you use. Share the modified CommonResults data type and the step types that use the CommonResults data type only with systems on which you are certain you will not deploy any conflicting changes to CommonResults. If you want to add additional results to particular steps, NI recommends using the Additional Results feature rather than modifying CommonResults.

Related concepts:

- [Type Versioning](#)
- [Managing Type Revisions](#)
- [Modifying Front-End Callbacks](#)
- [Type Concepts](#)

Path

Use the Path standard data type to include hyperlinks in supported versions of ATML, XML, and HTML reports.

NI recommends that you use relative paths, which can be relative to the sequence file that contains them or one of the TestStand search directories. Relative paths allow TestStand to find files even when you install or copy the files to a location on the another computer that is different from the location of the files on the original computer.

When you use the TestStand Deployment Utility to create a deployable image of a test system, the utility attempts to replace absolute file path references to code module files with relative file path references.



Note Absolute paths can cause errors in deployment and multiple-developer scenarios because directory structures frequently vary among systems. Absolute paths also increase the maintenance burden if you want to

move files within the TestStand system.

Related concepts:

- [Including Hyperlinks in Reports](#)
- [Search Directories](#)
- [Deploying TestStand Systems](#)
- [Creating a Deployable Image](#)
- [Processing TestStand Sequence Files for Deployment](#)
- [Including Hyperlinks in a Report - TDMS File](#)

Expression

Use the Expression standard data type to store an expression as a string so TestStand can locate expression values saved in variables and step properties when editing or analyzing sequence files.

Related concepts:

- [Expressions](#)

NI_TDMSReference

Use the NI_TDMSReference standard data type to identify references to NI Technical Data Management Streaming (.tdms) files in the TestStand `ResultList` local variable so you can log .tdms data in reports and in databases. Visit ni.com/tdms for more information about the .tdms file format.

The NI_TDMSReference data type includes the following subproperties. The report includes references to these subproperties only when you define values for the subproperties.

- **File**—Absolute path to the .tdms file you want to log in the report or database. TestStand automatically creates a `TestStand.Hyperlink` attribute set to `True` to include a hyperlink to the file in supported versions of ATML, XML, and HTML reports. If the value of the `Hyperlink` attribute is `False`, the report displays the path as a string, not a link. You can open links to the .tdms file

directly from supported versions of ATML, XML, and HTML reports and view information about the `.tdms` file in text reports.

- **Plugin**—Name of the DataPlugin that generated the file.
- **ChannelGroup**—Name of the channel group in which the data is recorded.
- **Channel**—Name of the channel in which the data is recorded.

When you use the XML Packaging Utility to distribute ATML, XML, and HTML reports that include links to `.tdms` files, the utility also distributes the target file of the link.

Related concepts:

- [Including Hyperlinks in Reports](#)
- [Including Hyperlinks in a Report - TDMS File](#)
- [Type Concepts](#)

Creating Custom Data Types

Complete the following steps to create a custom data type.

1. On the Types pane of the Types window, expand the **Custom Data Types** section.
2. Right-click and select **Insert Custom Data Type** from the context menu. You can also use the Copy and Paste context menu items to copy and rename an existing data type.
3. Select the data type you want from the submenu. When you select the Container type from the submenu, TestStand creates the data type without any fields in which you can insert additional data types.
4. Right-click the new data type and select **Properties** from the context menu to launch the Type Properties dialog box, in which you can specify the version number of the type and how to pass the data type to LabVIEW, LabWindows/CVI, and .NET code modules.

When you create new data types, use unique names to avoid conflicts with the default names TestStand uses and names other companies and users create. For example, begin new custom data type names with a unique ID, such as a company prefix.

Related concepts:

- [Creating Instances of Data Types](#)

- [Type Palette Files](#)

Related information:

- [Types Window](#)

Property Settings Specific to Data Types

TestStand data type properties include some settings that are specific to types. You can examine and modify the values of these settings in the Types window. Right-click a data type and select **Properties** from the context menu to launch the Type Properties dialog box.

Custom Step Types

You can create custom step types to meet the needs of an application. Custom step types differ from the step templates you store in the Templates list on the Insertion Palette, as the following table describes:

Item	Description	Effect of Change Made
Custom step types	Define standard functionality for a class of steps.	Changes can affect step instances previously inserted into sequences.
Step templates	Preconfigured instances of step types you typically use, such as calls to frequently used code modules.	Changes do not affect steps previously inserted into sequences.

Creating Custom Step Types

Complete the following steps to create a custom step type.

1. On the Types pane of the Types window, expand the **Step Types** section.
2. Right-click and select **Insert Step Type** from the context menu. You can also use the Copy and Paste context menu items to copy and rename an existing step. When you create new step types, use unique names to avoid conflicts with the default names TestStand uses. For example, begin new custom step type names with a

unique ID, such as a company prefix.

3. Right-click the new step and select **Properties** from the context menu to launch the Step Type Properties dialog box.
4. Click the Menu tab and specify the menu item name for the new step.
5. Click the General tab and specify the default name for new steps you create from the new type and specify the description expression for those steps.
6. Click the Substeps tab and select an adapter.
7. Click **Add** and select the type of step to create substeps. Use the Substep Info section on the Substeps tab to specify the menu item name of Edit steps.



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

Properties Common to All Step Types

TestStand defines multiple built-in step type properties common to all step types. Right-click a step and select **Properties** from the context menu to launch the Step Type Properties dialog box. You can examine and modify the values of the built-in step type properties in the Types window.



Note Changes to the default values do not automatically propagate to existing steps of this type.

The class step type properties exist only in the step type itself. TestStand uses the class step type properties to define how the step type works for all step instances. Step instances do not contain copies of the class step type properties.

The instance step type properties exist in each step instance. Each step you create with the step type includes a copy of the instance step type properties. TestStand uses the default value you specify for a step type property as the initial value of the property in each new step you create.

After you create a step, you can change the values of the properties for a step type instance, but these changes do not propagate to other step type instances. When you create a custom step type, you can prevent users from changing the values of specific

instance step type properties in the steps they create. For example, you can use the Edit substep of a step type to set the Status Expression for the step, in which case you do not want users to explicitly change the Status Expression value. Some of the built-in step types, such as the Numeric Limit Test and the String Value Test, prevent you from changing the value of the instance step type properties.

Custom Properties of Step Types

You can add any number of custom properties in a step type you create. Each step you create using the step type includes the custom properties you create.

1. On the Types pane of the Types window, expand the step type, right-click, select **Insert Field** from the context menu, and select a data type to add fields to a step type.
2. Right-click the field and use the context menu to cut, copy, paste, delete, and rename fields.

Configuring the Adapter Icon Color for a Custom Step Type

Configure the adapter icon color for a custom step type to enable users to distinguish between step types in the Step Types list and in a sequence.

1. On the Types pane of the Types window, locate and expand the step type.
2. Right-click and select **Insert Field** from the context menu, select **String**, and name it `Category`.
3. Click the Values column and enter one of the following values:
 - `Action`—sets the icon color to green.
 - `Test`—sets the icon color to magenta.
4. Hide the property on step instances. Step instances do not require the property.
 - a. Double-click `Category` to open the Type Properties dialog box.
 - b. Click **Advanced**, then select **Flags**.
 - c. Select **PropFlags_ExcludeFromCopy**.
 - d. (Optional) Select **PropFlags_Hidden** and **PropFlags_HiddenInTypes** to hide the property in the Types pane.

Related concepts:

- [Creating Custom Step Types](#)
- [Custom Properties of Step Types](#)

Using Substeps

Substeps use substep code modules to define standard actions, other than calling the step code module, TestStand performs for all instances of the step type. Use the Substeps tab of the Step Type Properties dialog box to specify Pre-Step, Post-Step, Edit, and Custom substeps for the step type.

After you add a substep to a step type, click the **Specify Module** button in the Substeps tab to configure the substep module call. For each step that uses the step type, TestStand calls the same substep modules with the same arguments. You cannot add or remove substeps or otherwise alter the substep module call the substep performs when you configure a step instance.

Although you can specify any number of substeps for a step type, the list of substeps is not a sequence, and substeps do not have preconditions, post actions, or other execution options. The order in which Pre- and Post-Step substeps execute is the only execution option you specify.

Pre-Step Substep

TestStand calls the Pre-Step substep before calling the step code module. For example, a Pre-Step substep might call a substep code module that retrieves measurement configuration parameters and stores those parameters in step properties the step code module uses.

Post-Step Substep

TestStand calls the Post-Step substep after calling the step code module. For example, a Post-Step substep might call a substep code module that compares the values the step code module stored in step properties against limit values the Edit substep stored in other step properties. You can have multiple Post-Step substeps that execute in order.

Edit Substep

TestStand calls an Edit substep when you select the substep menu item from the Steps pane context menu. On the Substeps tab, select the Edit substep and click the **Rename** button to specify the name of the substep menu item and the caption of the button on the Step Type Edit tab of the Step Setting pane. If the Edit substep modifies any step properties, increment the change count of the sequence file by calling the `PropertyObjectFile.IncChangeCount` method so an editing application can determine that the file changed and refresh itself.

The Edit substep typically calls a substep code module that launches a dialog box in which you can edit the values of the custom step properties. For example, an Edit substep might launch a dialog box in which you specify the high and low limits for a test. The Edit substep might then store the high and low limit values as step properties.

Dialog boxes the Edit substep launches must be modal. Refer to the <TestStand Public>\Examples\Fundamentals\Launching a Modal Dialog directory for LabVIEW and Microsoft Foundation Class (MFC) examples of modal dialog boxes.



Note You can initialize threads within TestStand executions to use the single-threaded apartment model or the multi-threaded apartment model. TestStand executes Edit substeps only in threads initialized using the single-threaded apartment model so the substep can open windows that contain ActiveX controls.



Note An alternative approach to handle user input for your custom step type is to create a custom step type edit tab that displays in the Step Settings pane of your step instance. This approach requires the development of a custom .NET control for integration into the Sequence Editor.

Custom Substeps

With the exception of substeps named `OnNewStep`, TestStand does not call custom substeps. Use the TestStand API to invoke a custom substep from a code module or user interface. You can create a custom substep named `OnNewStep` for TestStand to call each time you create a new step of that type. For example, the built-in If step type

uses an `OnNewStep` substep to insert a matching End step. The `OnNewStep` substep can cancel the insertion of the step by setting the `Step.CancelCurrentExecution` property to `True`.

The `<TestStand>\Components\StepTypes` directory includes source code for many of the substep modules the built-in step types use. To modify the installed step types or to create a new step type, copy the step type source code from the `<TestStand>\Components\StepTypes` directory to the `<TestStand Public>\Components\StepTypes` directory and make changes to the copy of the source code.

Related concepts:

- [TestStand Directory Structure](#)
- [Launching a Modal Dialog](#)
- [TestStand ActiveX API Overview](#)
- [Built-In Step Types](#)
- [Copying Read-Only Files to Modify](#)

Multiple Code Templates per Step Type

You can specify more than one code template for a step type. For example, you might want to have code templates that contain example code for conducting the same type of tests with different types of instruments or data acquisition boards.

When a step type has multiple code templates and you click the **Create Code** button on the Module tab of the Step Settings pane, TestStand prompts you to select from a list of templates or uses the template you selected on the Module tab when it exists.

Creating and Customizing Code Template Files

Click the **Create** button on the Code Templates tab of the Step Type Properties dialog box to launch the Create Code Templates dialog box.

TestStand copies the files for the existing code template you select into a new subdirectory in the `<TestStand Public>\CodeTemplates` directory based on the code template name you specified in the Create Code Templates dialog box. You can then

customize the code template files in the new <TestStand Public>\CodeTemplates directory.

For example, you can include example code that shows users how to access the custom properties of the step. For most environments, you can add a value parameter to pass the information from TestStand. You can also show how to obtain the high- and low-limit properties in a LabVIEW or LabWindows/CVI code template for a Numeric Limit Test step by customizing the prototype for the code module to specify the high and low limits as value parameters. As another example, you might want to show how to return a measurement value from a code module. For the LabVIEW, LabWindows/CVI, and C/C++ DLL Adapters, you can customize the prototype in the code template by specifying the measurement as a reference parameter.

Related concepts:

- [TestStand Directory Structure](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Programming with the TestStand API in LabWindows/CVI](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)

Template Files for Different Development Environments

Because different module adapters require different types of code modules, code templates typically correspond to a particular programming language in a specific development environment. The <TestStand>\CodeTemplates directory includes the default code templates for each development environment, as the following table describes.

Subdirectory Name	Template Description
Default_Template	Legacy default template
DefaultC++.NET	Default template for C++ in Microsoft Visual Studio
DefaultCSharp.NET	Default template for C# in Visual Studio
DefaultCVI	Default template for C in LabWindows/CVI
DefaultHTB72_Template	Default template for HTBasic 7.2
DefaultHTB80_Template	Default template for HTBasic 8.0

Subdirectory Name	Template Description
DefaultLabVIEW	Default template for LabVIEW
DefaultVB.NET	Default template for Microsoft Visual Basic .NET
DefaultVC++_Template	Default template for C++ in Visual Studio

Each subdirectory includes the source file for the module adapter and a `.ini` file that contains parameter information and a description string TestStand displays for the code template. TestStand uses the directory names from the `<TestStand>\CodeTemplates` or `<TestStand Public>\CodeTemplates` directory as the code template name to display in the Code Templates tab of the Step Type Properties dialog box.

Code templates for the LabVIEW, LabWindows/CVI, and C/C++ DLL Adapters can have any number of parameters compatible with the data types you can specify on the Module tab of the Step Settings pane for those adapters.

When TestStand uses a code template for a DLL to create skeleton code, it compares the parameter list in the source file to the parameter information on the Module tab. When these two sources of information do not match, TestStand prompts you to select which prototype to use for the skeleton code. When you use the prototype from the template source file, TestStand updates the Module tab to match the prototype in the template source file. However, the template source file does not contain sufficient information for TestStand to update the Value controls for the parameters on the Module tab. Use the Parameter Name/Value Mappings section of the Edit Code Template dialog box to specify entries for TestStand to place in the Value controls. TestStand stores the parameter values in the `.ini` file in the template subdirectory.

Legacy Code Templates

In TestStand 3.5 or earlier, code template directories contain source files for multiple development environments. For example, a legacy code template directory might include one `.c` file for the LabWindows/CVI Adapter and multiple VIs for the LabVIEW Adapter, where each VI corresponds to the different combinations of parameter options users can set in the Edit LabVIEW VI Call dialog box. TestStand includes these legacy code templates to provide compatibility with previous versions of TestStand. The `[TemplateType]` section of the `config.ini` file in each code template

directory includes `Type = "Legacy"` for legacy code templates.

Legacy Code Templates for LabVIEW

Legacy code templates for the LabVIEW Adapter always specify **Test Data** and **Error Out** clusters as parameters. The VIs for each LabVIEW Adapter legacy code template specify various combinations of the **Input Buffer**, **Invocation Info**, and **Sequence Context** parameters. When TestStand uses a legacy LabVIEW template VI to create skeleton code, it selects the correct VI to use according to the current settings in the Optional Parameters dialog box in TestStand 3.5 or earlier.

Legacy Code Templates for LabWindows/CVI

Legacy code templates for the LabWindows/CVI Adapter always specify two parameters—a pointer to a **tTestData** structure and a pointer to a **tTestError** structure. When TestStand uses a legacy LabWindows/CVI template module to create skeleton code, it validates the function prototype in the template module against this requirement. TestStand reports an error when the prototype is incorrect.

Related concepts:

- [TestStand Directory Structure](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Programming with the TestStand API in LabWindows/CVI](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)

Managing Step Type Version Compatibility

When you modify custom step types, consider the following caveats so you can avoid making changes that might jeopardize compatibility.

- Ensure that previously configured steps behave properly when you execute the steps using the modified custom step type.
- Ensure that new step instances based on the modified step type behave properly when you save a sequence file to an earlier version of TestStand that uses the original custom step type.
- Do not rename custom step type properties or change the functionality of existing

properties. For example, when you have an existing property that performs a specified task and you later decide you want the property to do something completely different in a future instance of the step type, you break compatibility.

- When you create new properties, provide default values that preserve the functionality of previously created steps.
- When you extend enumerated values, do not change the functionality of previously used values.

Adding Step Types to Surround Selection With Context Menu

Complete the following steps to add a custom step type to the Surround Selection With context menu of the Steps pane and to add an end step after the last step in the contiguous block of steps you selected.

1. Complete the following steps to add the custom step type to the Surround Selection With context menu.
 - a. Right-click the custom step type and select **Properties** to launch the Step Type Properties dialog box.
 - b. Click the **Advanced** button and select **Block Structure** from the drop-down menu to launch the Block Structure dialog box.
 - c. Enable the **Can Surround a Selection of Steps** option to include the custom step type in the Surround Selection With context menu.
2. Create an OnNewStep substep to execute when you insert an instance of the custom step type.
3. Create a code module for the OnNewStep substep to check the value of the Boolean `SequenceContext.RunState.ShouldEncapsulate` property to determine where to add a corresponding end step. When you use the Insert Step menu or the Insertion Palette to insert a new step, the `SequenceContext.RunState.ShouldEncapsulate` property does not exist, and the `SequenceContext.SelectedSteps` property contains the newly inserted step. When you use the Surround Selection With menu to surround a set of selected steps with corresponding begin and end steps, the `SequenceContext.RunState.ShouldEncapsulate` property is `True`, and the `SequenceContext.SelectedSteps` property contains the set of selected steps. In the code module for the OnNewStep substep, verify that the `SequenceContext.RunState.ShouldEncapsulate` Boolean property exists and is `True`. When the

`SequenceContext.RunState.ShouldEncapsulate` property is `True`, you must add the corresponding end step after the last step of the selected steps. When the `SequenceContext.RunState.ShouldEncapsulate` property is `False` or does not exist, you can add the end step after the newly inserted step or in another location you specify.

Related concepts:

- [Using Substeps](#)

Customizing Process Models and Callbacks

You can customize the default process models and callback sequences. Customize callbacks to implement custom functionality specific to unit under test (UUT) models. Customize process models to implement functionality that is standard throughout an organization and applies to all or most UUTs. Use callbacks to implement functionality you might change. Use process models to implement functionality you are unlikely to change. You can extend the functionality of existing process models by creating or customizing process model plug-ins.



Note If you modify any process model or process model plug-in types, open and save all the process models and process model plug-ins that use the modified types to ensure you avoid a type conflict. To modify or save the process models, you must first copy all the files from the `<TestStand>\Components\Models\TestStandModels` directory to the `<TestStand Public>\Components\Models\TestStandModels` directory and modify or save the files in the `<TestStand Public>\Components\Models\TestStandModels` directory. The files in the `<TestStand Public>\Components\Models\TestStandModels` directory override the files in the `<TestStand>\Components\Models\TestStandModels` directory. Similarly, you must copy any process model plug-in files from the `<TestStand>\Components\Models\ModelPlugins` directory to the `<TestStand Public>\Components\Models\ModelPlugins` directory before modifying the files.

Related concepts:

- [Process Model Architecture](#)
- [Callback Sequences](#)
- [Process Model Plug-In Architecture](#)
- [TestStand Directory Structure](#)

Modifying Process Model Sequence Files

You must modify the process model sequence files directly to make changes that apply wherever TestStand uses the process model.

TestStand installs the SequentialModel.seq, ParallelModel.seq, and BatchModel.seq process model sequence files and supporting files in the <TestStand>\Components\Models\TestStandModels directory. To modify the installed process model or to create a new process model, copy all the process model files from the <TestStand>\Components\Models\TestStandModels directory to <TestStand Public>\Components\Models and make changes to the copy.

In addition to editing process model sequence files, you can convert normal sequence files to process model sequence files. Complete the following steps to specify a sequence file as a process model sequence file.

1. Select the sequence file and select **Edit»Sequence File Properties**.
2. In the Sequence File Properties dialog box, click the Advanced tab.
3. Select **Model** from the Type ring control.
4. Click **OK**.

Although you edit a process model sequence file in a regular Sequence File window, the file includes Model entry points and Model callbacks. TestStand maintains special properties for entry point and callback sequences, and you can specify the values of these properties when you edit the sequences in a process model file.

When you access the Sequence Properties dialog box for any sequence in a model file, the dialog box contains a Model tab you use to specify the sequence as a normal sequence, a callback sequence, or an entry point sequence.

Related concepts:

- [Sequential Process Model](#)
- [Parallel Process Model](#)
- [Batch Process Model](#)
- [TestStand Directory Structure](#)
- [Copying Read-Only Files to Modify](#)
- [Entry Point Sequences](#)
- [Using Callback Sequences to Modify Process Models](#)
- [Normal Sequences](#)

Normal Sequences

A normal sequence is any sequence other than a callback or an entry point. In process model files, use normal sequences as Utility subsequences that entry points or callbacks call. When you select Normal from the Type ring control on the Model tab of the Sequence Properties dialog box, the Model tab does not include any other options.

Related concepts:

- [Process Model Architecture](#)

Using Callback Sequences to Modify Process Models

Model callbacks are sequences entry point sequences call and client sequence files can override. Use Model callbacks to customize the behavior of a process model for each client sequence file that uses the process model. By defining one or more Model callbacks in a process model file, you specify the set of process model operations you can customize from a client sequence file.

Complete the following steps to define a Model callback.

1. Add a sequence to the process model file.
2. Select **Edit»Sequence Properties** to launch the Sequence Properties dialog box.
3. Click the Model tab.
4. Select **Callback** from the Type ring control.
5. Click **OK**.
6. Call the new sequence you just created from the process model.

You can override a callback in the process model sequence file by using the Sequence

File Callbacks dialog box to create a sequence with the same name but different functionality in the client sequence file. Select **Edit»Sequence File Callbacks** to launch the Sequence File Callbacks dialog box.

Some Model callbacks, such as the TestReport callback in the default process model, are sufficient for handling specific types of operations. Other Model callbacks are placeholders you override with sequences in the client sequence file. For example, the `MainSequence` callback in the default process model file is a placeholder for the `MainSequence` callback you create in the client sequence file.

A primary process model file can directly call model callbacks in a secondary process model file. At run time, if the client sequence file of the primary file implements a callback defined in the secondary process model file, TestStand invokes the callback in the client sequence file, even if the primary process model file does not define the callback. You must add a copy of the callback to the primary model file for the callback to appear in the Sequence File Callbacks dialog box for the client sequence file.



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

Related concepts:

- [Model Callbacks in the Batch Process Model](#)
- [TestStand Directory Structure](#)

Entry Point Sequences

You can invoke Execution entry point sequences and Configuration entry point sequences from the TestStand Sequence Editor or User Interface menus to run client files or to configure model settings. Each entry point is a sequence in the process model file.

Execution Entry Points

Execution entry points run test programs typically by calling the `MainSequence` callback in the client sequence file. The Sequential, Parallel, and Batch process models contain the following Execution entry points:

- **Test UUTs**—Tests and identifies multiple UUTs or batches of UUTs in a loop.
- **Single Pass**—Tests one UUT or a single batch of UUTs without identifying the UUTs.

By default, the Execute menu lists Execution entry points only when the active window contains a sequence file that uses the process model.

Configuration entry points configure a feature of the process model and usually save the configuration information in a `.ini` file in the `<TestStand Application Data>\Cfg` directory. The process models contain the following Configuration entry points:

- **Configure Result Processing**—Launches the Result Processing dialog box, in which you enable or disable the installed built-in and custom result processing plug-ins in the active result processing configuration and also configure how the plug-ins process test results. Use the built-in reporting, database, and offline results plug-ins to generate reports, log data to databases, and store results in compact offline result files for processing later, respectively. By default, you can configure the following result processors:
 - **Database**—Configure a database results processor to enable UUT result logging and configure the schema for mapping TestStand results to database tables and columns.
 - **Report**—Configure a reports results processor to enable UUT report generation and configure the report type and contents of the report files.
 - **Offline Results File**—Configure an offline results processor to enable storing of TestStand results in a compact raw result format (`.tsr`) for offline processing at a time other than when sequence execution occurs or on a different computer, and to configure the location of the offline results files.
- **Configure Model Options**—Launches the Model Options dialog box, in which you configure the number of test sockets and other process model-related options.

Configuration Entry Points

By default, the Configure menu lists the Configuration entry points.

Related concepts:

- [Model Callbacks in the Batch Process Model](#)
- [Sequential Process Model](#)
- [Parallel Process Model](#)
- [Batch Process Model](#)
- [TestStand Directory Structure](#)

Modifying Callbacks

TestStand includes Engine callback and Front-End callback sequences you can customize to meet specific needs.

Related concepts:

- [Modifying Engine Callbacks](#)
- [Modifying Front-End Callbacks](#)
- [Overriding PreBatch and PostBatch Model Callbacks](#)
- [Overriding PreUUT and PostUUT Batch Model Callbacks](#)
- [Overriding PreUUT and PostUUT Parallel Model Callbacks](#)

Modifying Engine Callbacks

The TestStand Engine invokes a set of Engine callbacks at specific points during execution. TestStand defines the set of Engine callbacks and the callback names because the engine controls the execution of steps and the loading and unloading of sequence files.

Use Engine callbacks to configure TestStand to call certain sequences at various points during a test, including before and after the execution of individual steps, before and after interactive executions, after loading a sequence file, or before unloading a sequence file.

TestStand categorizes Engine callbacks according to the file in which the callback appears. You can define Engine callbacks in sequence files, process model files, and in `StationCallbacks.seq` in the <TestStand Public>\Components\Callbacks\Station directory.



Note TestStand does not predefine any Station Engine callbacks in `StationCallbacks.seq` in the `<TestStand>\Components\Callbacks\Station` directory but might in a future version of TestStand.

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

You can use Engine callbacks in the following ways:

- Use the `SequenceFileLoad` callback to ensure that you configure external resources the sequence file uses only once before you execute the sequence. Usually, you initialize devices a sequence requires by creating steps in the Setup step group for the sequence. However, when you call the sequence repeatedly, you can move the Setup steps into a `SequenceFileLoad` callback for the subsequence file so that the steps run only when the sequence file loads.
- Use the `StationPreStep` and `StationPostStep` callbacks to accumulate statistics on all steps that execute on the test station. You can inspect the name and types of steps that accumulate data on specific steps.

Related concepts:

- [Engine Callbacks](#)
- [TestStand Directory Structure](#)
- [List of Engine Callbacks](#)
- [Caveats for Using Engine Callbacks](#)
- [Overriding PreBatch and PostBatch Model Callbacks](#)
- [Overriding PreUUT and PostUUT Batch Model Callbacks](#)
- [Overriding PreUUT and PostUUT Parallel Model Callbacks](#)

List of Engine Callbacks

The following table lists the Engine callbacks TestStand defines, indicates where you must define the callback, and specifies when the engine calls the callback.

Engine Callback	Where You Define the Callback	When the Engine Calls the Callback
SequenceFilePreStep	Any sequence file	Before the TestStand Engine executes each step in the sequence file.
SequenceFilePostStep	Any sequence file	After the engine executes each step in the sequence file. The engine does not call this callback when a step errors.
SequenceFilePreInteractive	Any sequence file	Before the engine begins an interactive execution of steps in the sequence file.
SequenceFilePostInteractive	Any sequence file	After the engine completes an interactive execution of steps in the sequence file.
SequenceFileLoad	Any sequence file	When the engine loads the sequence file into memory.
SequenceFileUnload	Any sequence file	When the engine unloads the sequence file from memory.
SequenceFilePostResultListEntry	Any sequence file	After the engine fills out the step result for a step in the sequence file.
SequenceFilePostResults	Any sequence file	After the engine collects a step result in the sequence file and the number of collected results or the time since collected results were last processed exceeds a threshold.
SequenceFilePostStepRuntimeError	Any sequence file	After a step in the sequence file generates a run-time error.
SequenceFilePostStepFailure	Any sequence file	After a step in the sequence fails.
ProcessModelPreStep	Process model file	Before the engine executes each step in any client sequence file the process model calls and each step in

Engine Callback	Where You Define the Callback	When the Engine Calls the Callback
		any resulting subsequence calls.
ProcessModelPostStep	Process model file	After the engine executes each step in any client sequence file the process model calls and each step in any resulting subsequence calls. The engine does not call this callback when a step errors.
ProcessModelPreInteractive	Process model file	Before the engine begins an interactive execution of steps in a client sequence file and steps in any resulting subsequence calls.
ProcessModelPostInteractive	Process model file	After the engine completes an interactive execution of steps in a client sequence file and steps in any resulting subsequence calls.
ProcessModelPostResultListEntry	Process model file	After the engine fills out the step result for a step in any client sequence file the process model calls or in any resulting subsequence calls.
ProcessModelPostResults	Process model file	After the engine collects a step result for a step in any client sequence file the process model calls or in any resulting subsequence calls and the number of collected results or the time since collected results were last processed exceeds a threshold.
ProcessModelPostStepRuntimeError	Process model file	After a step generates a run-time error when the step is in a client sequence file the process model calls or in any

Engine Callback	Where You Define the Callback	When the Engine Calls the Callback
		resulting subsequence calls.
ProcessModelPostStepFailure	Process model file	After a step fails when the step is in a client sequence file the process model calls or in any resulting subsequence calls.
StationPreStep	StationCallbacks.seq	Before the engine executes each step in any sequence file.
StationPostStep	StationCallbacks.seq	After the engine executes each step in any sequence file. The engine does not call this callback when a step errors.
StationPreInteractive	StationCallbacks.seq	Before the engine begins any interactive execution.
StationPostInteractive	StationCallbacks.seq	After the engine completes any interactive execution.
StationPostResultListEntry	StationCallbacks.seq	After the engine fills out the step result for a step in any sequence file.
StationPostResults	StationCallbacks.seq	After the engine collects a step result for a step in any sequence file and the number of collected results or the time since collected results were last processed exceeds a threshold.
StationPostStepRuntimeError	StationCallbacks.seq	After any step generates a run-time error.
StationPostStepFailure	StationCallbacks.seq	After any step fails.



Note The threshold is in terms of the number of results the current thread has accumulated and the amount of time since the current thread last called its PostResults callbacks. For more information about the conditions that trigger PostResults callbacks, refer to `Execution.PostResultsCallbackOptions`,

Execution.PostResultsCallbackInterval,
Execution.MaximumResultsPerPostResultsCallback, and
Thread.FlushPostResults.

Related concepts:

- [Engine Callbacks](#)
- [Caveats for Using Engine Callbacks](#)
- [Overriding PreBatch and PostBatch Model Callbacks](#)
- [Overriding PreUUT and PostUUT Batch Model Callbacks](#)
- [Overriding PreUUT and PostUUT Parallel Model Callbacks](#)

Caveats for Using Engine Callbacks

Consider the following issues when you define Engine callbacks:

- Typically, you do not define SequenceFile<xxx> callbacks in a process model file because these types of callbacks apply only to the steps in the process model file.
- Do not define a SequenceFileLoad or SequenceFileUnload callback in the `StationCallbacks.seq` because TestStand does not call these callbacks.
- When a callback sequence is empty, TestStand does not invoke the Engine callback.
- Process models use the Execution.EnableCallback method to disable the ProcessModelPostResults callback when the model does not need to process results on-the-fly for report generation, database logging, offline result file generation, or for any other result processing plug-in.
- TestStand calls other Engine callbacks only when executing the SequenceFileLoad or SequenceFileUnload Engine callbacks. TestStand does not call Engine callbacks when executing the other Engine callbacks.
- When TestStand reloads a sequence file after the file is modified on disk, TestStand can call the SequenceFileLoad callback in the new file without calling the SequenceFileUnload callback in the original file.
- TestStand passes the ProcessModelPostResults, SequenceFilePostResults, and StationPostResults callbacks an extra result for each Sequence Call step. The extra result, called a provisional result, accumulates in the executing thread before the thread calls the sequence the step specifies. Provisional results do not appear in the result list and TestStand does not pass provisional results to the

PostResultListEntry callbacks. A provisional result includes a `TS.Provisional` Boolean property that is set to `True`. If the `ProcessModelPostResults`, `SequenceFilePostResults`, or `StationPostResults` callback does not need provisional results, it can use the `TS.Provisional` field to identify and ignore them.

Related concepts:

- [List of Engine Callbacks](#)
- [Step Results](#)

Modifying Front-End Callbacks

Front-End callbacks are sequences in the `FrontEndCallbacks.seq` file multiple user interface applications can call to share the same implementation for a specific operation. The `FrontEndCallback.seq` file TestStand installs in the `<TestStand>\Components\Callbacks\FrontEnd` directory contains one `LoginLogout` Front-End callback. The TestStand Sequence Editor and default user interfaces call the `LoginLogout` callback.

Use Front-End callbacks to implement operations so you can modify a Front-End callback without modifying the source code for the user interfaces or rebuilding the executables for the user interfaces. For example, to change how various user interfaces perform the login procedure, modify only the `LoginLogout` sequence in `FrontEndCallbacks.seq`.

To modify the default implementation of the Front-End callback or to create new Front-End callbacks, copy the `FrontEndCallbacks.seq` file from the `<TestStand>\Components\Callbacks\FrontEnd` directory to the `<TestStand Public>\Components\Callbacks\FrontEnd` directory and make any changes to the copy of the file. You can use functions in the TestStand API to invoke the modified Front-End callback from each user interface application you create. However, because you cannot edit the source for the sequence editor, you cannot make the sequence editor call new Front-End callbacks you create.

Related concepts:

- [Callback Sequences](#)

- [TestStand Directory Structure](#)
- [Copying Read-Only Files to Modify](#)
- [TestStand ActiveX API Overview](#)

Creating Custom User Interfaces

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

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

- Example User Interfaces
- Writing an Application with the TestStand UI Controls
- TestStand ActiveX API Overview
- Core UI Classes, Properties, Methods, and Events
- Core API Classes, Properties, and Methods
- Using TestStand UI Controls in LabVIEW
- Using TestStand UI Controls in LabWindows/CVI

Refer to the following TestStand documents for more information about creating and using custom user interfaces with LabVIEW or LabWindows/CVI.

- Programming with the TestStand API in LabVIEW
- Programming with the TestStand API in LabWindows/CVI
- The NI TestStand User Interface Controls Reference Poster

Refer to the following sources for general instructions for creating a user interface application with an environment other than LabVIEW or LabWindows/CVI.

- Programming with the TestStand API in Different Languages

Related concepts:

- [Example User Interfaces](#)
- [Writing an Application with the TestStand User Interface Controls](#)
- [TestStand ActiveX API Overview](#)
- [Using TestStand UI Controls in LabVIEW](#)
- [Using TestStand UI Controls in LabWindows/CVI](#)

- [Programming with the TestStand API in LabVIEW](#)
- [Programming with the TestStand API in LabWindows/CVI](#)
- [Programming with the TestStand API in Different Languages](#)

TestStand UI Controls

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



Note NI strongly recommends that you use the TestStand UI Controls to develop user interface applications.

User interfaces that use the TestStand UI Controls typically perform the following basic operations:

- Configure connections, commands, and other control settings
- Register to handle events the controls generate
- Start TestStand
- Wait in a main event loop until you close the application
- Shut down TestStand

User interfaces can also include a menu bar that contains non-TestStand items and items that invoke TestStand commands.

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

- You do not need to create the TestStand Engine. Use the `ApplicationMgr.GetEngine` method to obtain the Engine object.
- When you call the `Engine.NewExecution` method to create an execution, the TestStand UI Controls recognize the new execution.
- When you call the `Engine.GetSequenceFileEx` method to load a sequence file, the TestStand UI Controls do not display the file you load. You must call the `ApplicationMgr.OpenSequenceFile` method to open and display a file in the user

interface.

- You can obtain sequence file and execution references from events or from the SequenceFiles and Executions collections.
- When you hold references to TestStand objects, release them in the handler for the ApplicationMgr.QueryShutdown event when the event handler does not cancel the shut down process.
- The TestStand UI Controls display only the active thread and do not support enabling the `ExecMask_TraceAllThreads` ExecutionMask constant.
- Do not couple user interfaces to specific sequences.
- Do not use instrumentation code in user interfaces.
- Use UI Messages to communicate information between code modules and user interfaces.

Related concepts:

- [Getting Started with User Interface Development](#)
- [Executions](#)
- [Setting the Preferred Execution System of LabVIEW VIs to Prevent Deadlock](#)
- [Writing an Application with the TestStand Engine API](#)

Using TestStand UI Controls in Different Environments

You can use the TestStand User Interface (UI) Controls in LabVIEW, LabWindows/CVI, Microsoft Visual Studio, and Microsoft Visual C++.

Related concepts:

- [TestStand UI Controls](#)
- [Using TestStand UI Controls in LabVIEW](#)
- [Using TestStand UI Controls in LabWindows/CVI](#)
- [Using TestStand UI Controls in Microsoft Visual Studio](#)
- [Using TestStand UI Controls in Microsoft Visual C++](#)

Using TestStand UI Controls in LabVIEW

You can create custom user interfaces and create user interfaces for other components, such as custom step types. Use the TestStand User Interface (UI) Controls on the **Controls»TestStand** palette in LabVIEW to develop a custom user interface application,

including custom sequence editors.

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

TestStand VIs and Functions

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

Use the TestStand VIs and functions for the following tasks:

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

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

TestStand Version Selector

The TestStand Version Selector installs the **Controls»TestStand** palette and the **Functions»TestStand** palette when it detects an installation of LabVIEW. If you have just installed a new version of LabVIEW, the palettes do not appear in LabVIEW until you launch the TestStand Engine, at which point the TestStand Version Selector detects the new version of LabVIEW and installs the palette files. You can also launch the TestStand Version Selector, select the current version of TestStand, and click the

Make Active button to re-activate the current version of TestStand and to re-install the palette files.



Note Refer to the Starting and Shutting Down TestStand from a LabVIEW User Interface topic for more information about properly starting and shutting down TestStand from LabVIEW applications.

Related concepts:

- [TestStand UI Controls](#)
- [TestStand Utility Functions Library](#)
- [Starting and Shutting Down TestStand from a .NET User Interface](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Starting and Shutting Down TestStand from a LabVIEW User Interface](#)

Configuring the TestStand UI Controls for a LabVIEW User Interface

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

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

Related concepts:

- [Example User Interfaces](#)
- [TestStand UI Controls](#)

Special Considerations for a LabVIEW Custom User Interface

Consider the following guidelines when running LabVIEW user interfaces:

- You must close all running LabVIEW user interfaces before you exit Microsoft Windows. When you shut down, restart, or log off of Windows while a user interface is running, the user interface cancels the operation and might exit with an error.
- When you run a LabVIEW user interface in the LabVIEW development system, you can call remote VIs only when the VI is the same or earlier version as the LabVIEW development system. TestStand does not support calling VIs on remote computers when you save a VI without the block diagram and the version of the VI is different than the active version of LabVIEW installed on the computer.
- By default, you can run only one copy of a LabVIEW-built executable at a time, which prevents the TestStand `/useexisting` command-line option from working. Add the `"allowmultipleinstances = TRUE"` option to the INI options file in the same directory as the LabVIEW-built executable to allow more than one copy to execute at a time.
- (LabVIEW 2013 SP1 or earlier) A LabVIEW user interface reports object leaks on shutdown when you pass a TestStand object as the **ActiveXData** parameter of a UIMessage in the UIMessage callback VI. LabVIEW does not release the object reference until LabVIEW unloads the callback VI. Visit ni.com/info and enter the Info Code 4H6DULT3 to access the NI KnowledgeBase article PropertyObjects Were Not Released Warning When Using LabVIEW for more information about using a dynamically called VI to ensure that LabVIEW releases the object.
- To prevent a LabVIEW user interface from hanging while running in LabVIEW, do not call subroutine VIs from inside a VI used as an ActiveX callback VI. Instead, change the execution priority of the VIs to a value other than subroutine.

Related concepts:

- [Calling VIs on Remote Computers](#)

Using TestStand UI Controls in LabWindows/CVI

You can create custom user interfaces and create user interfaces for other components, such as custom step types. Use the TestStand User Interface (UI) Controls in the LabWindows/CVI development environment to develop a custom user interface application, including custom sequence editors.

Creating and Configuring ActiveX Controls

In the LabWindows/CVI User Interface Editor, select **Create»ActiveX** and select a user interface control beginning with `TestStand UI` to add a TestStand UI Control to a panel. Double-click the control to launch the standard LabWindows/CVI Edit Control dialog box, in which you can configure the control. Right-click the control and select **Properties** from the context menu to open the property pages the UI control supports.

Programming with ActiveX Controls

To access the methods, properties, and events specific to an ActiveX control, you must use the ActiveX driver for the control. The TestStand UI Controls driver and additional support instrument drivers are located in the `<TestStand>\API\CVI` directory. TestStand copies these instrument driver files (`.h`, `.c`, `.fp`, and `.obj`) to the `<National Instruments>\Shared\CVI\instr\TestStand\API` directory. The TestStand Version Selector copies API files for earlier versions of LabWindows/CVI to the appropriate locations.

Add the following function panel files to the LabWindows/CVI project for a TestStand application and add the corresponding header files in the source code files that use the API:

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

modal to TestStand applications, and checking if an execution that calls a code module has stopped.

- **TestStand API** (`tsapicvi.fp`) —Provides low-level access to TestStand objects.

For each interface the ActiveX control supports, the driver contains a function you can use to programmatically create an instance of the ActiveX control. The ActiveX driver also includes functions you can use to register callback functions for receiving events the control defines.

When you store ActiveX controls in `.uir` files, you do not need to use the creation functions the driver includes because the control is created when you load the panel from the file using the `LoadPanel` function. You identify the control in subsequent calls to User Interface Library functions with the constant name you assigned to the control in the User Interface Editor.

When you use other functions in the driver, you must identify the control with a unique object handle that LabWindows/CVI then associates with the control. You obtain this handle when you call the `GetObjHandleFromActiveXCtrl` function using the constant name for the control. This handle is cached in the control, and you do not need to discard the handle explicitly.

LabWindows/CVI requires you to initialize a thread as apartment-threaded before you can use ActiveX controls in a program. When you do not initialize the thread before creating an ActiveX control or before loading a panel containing an ActiveX control from a `.uir` file, LabWindows/CVI automatically initializes the thread to apartment-threaded. When you use the

`CA_InitActiveXThreadStyleForCurrentThread` function to initialize the thread yourself, you must use `COINIT_APARTMENTTHREADED` as the threading model.

Include Files and Instrument Search Directories

The TestStand Version Selector automatically adds the additional required API files to shared directories for the installed versions of LabWindows/CVI. If you have just installed a new version of LabWindows/CVI, the API files are not available until after you launch the TestStand Engine, at which point the TestStand Version Selector detects the new version of LabWindows/CVI and copies the required API files. You can also launch the TestStand Version Selector, select the current version of TestStand, and

click the **Make Active** button to re-activate the current version of TestStand and to re-copy the API files.



Note Refer to the Starting and Shutting Down TestStand from a LabWindows/CVI User Interface topic for more information about properly starting and shutting down TestStand from .NET Framework applications.

Related concepts:

- [TestStand UI Controls](#)
- [TestStand Directory Structure](#)
- [Starting and Shutting Down TestStand from a LabWindows/CVI User Interface](#)
- [Launching an MFC Dialog with ActiveX Controls](#)
- [Programming with the TestStand API in LabWindows/CVI](#)

Binding a TestStand Custom User Interface to a Side-by-Side Version of the LabWindows/CVI RTE

You can use the following options in LabWindows/CVI 2012 or later to bind a user interface executable to a specific, side-by-side version of the LabWindows/CVI RTE:

- **<version> side-by-side for executable only**—Use this option to use the side-by-side version of the LabWindows/CVI RTE only for the user interface executable. This option does not affect which version of the LabWindows/CVI RTE used by DLL modules that the LabWindows/CVI Adapter or the C/C++ DLL Adapter call. Those DLL modules use the version of the LabWindows/CVI RTE to which they were bound.
- **<version> side-by-side for entire application**—Use this option to use the side-by-side version of the LabWindows/CVI RTE for the user interface executable and for DLL modules bound to the default LabWindows/CVI RTE the LabWindows/CVI Adapter and C/C++ DLL Adapter call.

Related concepts:

- [Choosing Between the Shared or Side-by-Side Version of the LabWindows/CVI Run-Time Engine](#)
- [Using Side-by-Side Versions of the LabWindows/CVI Run-Time Engine with TestStand](#)

Configuring the TestStand UI Controls for a LabWindows/CVI User Interface

The following table lists the functions in the example user interface files that demonstrate configuring connections, commands, and other settings for the TestStand User Interface (UI) Controls.

Source Files	Functions
<TestStand Public>\UserInterfaces\ Simple\CVI\Source Code\ TestExec.c	SetupActiveXControls
<TestStand Public>\UserInterfaces\ Full-Featured\CVI\ Source Code\TestExec.c	GetActiveXControlHandlesRegisterActiveXEventCallbacks

Related concepts:

- [Example User Interfaces](#)
- [TestStand UI Controls](#)
- [TestStand Directory Structure](#)

Handling Variants in LabWindows/CVI User Interfaces

Several functions in the TestStand API return variants. If the return value of these functions is a reference to an ActiveX/COM object, you must cast these variants to a

valid handle type. To do this properly in LabWindows/CVI, you must create a temporary variable to hold the value and then convert the value to the correct type of handle.

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

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

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

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

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

```
HRESULT CVICALLBACK ApplicationMgr_OnUserMessage(CAObjHandle caServerObjHandle,
void
*caCallbackData, TSUIObj_UIMessage uiMsg)
{
    int error = 0;
    ErrMsg errMsg = "";
    ERRORINFO errorInfo;
    LPUNKNOWN tempPropObject = 0;
    CAObjHandle PropObject = 0;
    tsErrChk(TS_UIMessageGetActiveXData(uiMsg, NULL, &tempPropObject));
    CA_CreateObjHandleFromInterface (tempPropObject, &IID_IUnknown, 0,
    LOCALE_NEUTRAL, 0, 0, &PropObject);
    //TODO: Implement your code here
Error:
    //free resources
    if (PropObject)
        CA_DiscardObjHandle(PropObject);
    return error;
}
```

Using TestStand UI Controls in Microsoft Visual Studio

To use the TestStand User Interface (UI) Controls in Microsoft Visual Studio, drag the

TestStand UI Controls from the TestStand tab on the Visual Studio Toolbox onto a form.

When you create a new project in a supported version of Visual Studio, select **Project»Project Name Properties**, click the **Build** tab, and select **x86** from the **Platform Target** ring control in the General section of the dialog box so the project can access the TestStand API and UI Controls on 64-bit versions of Microsoft Windows.

Visit ni.com/info and enter the Info Code `tsuiivs2010` to access the NI KnowledgeBase article 5NK6NKXU, Why do I Receive an Error when Using Visual Studio 2010 to Build a TestStand User Interface?, for information about building TestStand user interfaces in Visual Studio 2010.

If the Visual Studio Toolbox window does not display the TestStand tab when you edit a form, or if the TestStand Interop assemblies do not appear in the Add References dialog box, exit all running copies of Visual Studio, launch the TestStand Version Selector, select the current version of TestStand, and click the **Make Active** button.

You must also add references to the TestStand Interop assemblies and the TestStand Utility (TSUtil) assembly to the project.

When you create a Multiple Document Interface (MDI) application with TestStand UI Controls on an MDI child form, Microsoft .NET resets the properties you programmatically set on the TestStand UI Controls to default values when you set the `MdiParent` property on the child form. .NET resets these properties because .NET destroys and recreates ActiveX controls on a form when you set the property on the form. To avoid this issue, set the TestStand control properties after you set the `MdiParent` property on the form or place all TestStand UI Controls and other ActiveX controls on a Panel control instead of directly on the form.



Note Refer to the Starting and Shutting Down TestStand from a .NET User Interface topic for more information about properly starting and shutting down TestStand from .NET applications.

Related concepts:

- [TestStand UI Controls](#)
- [Adding Assembly References in Microsoft Visual Studio](#)

- [TestStand Utility Functions Library](#)
- [Starting and Shutting Down TestStand from a .NET User Interface](#)
- [Programming with the TestStand API in C#](#)

Using TestStand UI Controls in Microsoft Visual C++

To use the TestStand User Interface (UI) Controls in Microsoft Visual C++, add the TestStand Utility (TSUtil) Functions Library to the project. The `TSUtilCPP.cpp` and `TSUtilCPP.h` files automatically import the type libraries for the TestStand API and the TestStand User Interface (UI) Controls.

You can view the header files the `#import` directive generates for the TestStand API type libraries by opening the `tsui.tlh`, `tsuisupp.tlh`, and `tsapi.tlh` files Visual C++ creates in the `Debug` or `Release` directory. The header files the `#import` directive generates define a C++ class for each object class in the TestStand API. The `I` prefix in class names denotes ActiveX controls and objects you can create without calling another class. The header files use macros to define a corresponding smart pointer class for each object class. Each smart pointer class uses the name of the corresponding class and adds a `Ptr` suffix. Typically, you use only smart pointer classes in an application because the smart pointer releases the reference to the object when the pointer is destroyed. For example, instead of using the `SequenceFile` class, use the `SequenceFilePtr` class.



Note NI recommends, in accordance with Microsoft guidelines, using the classes the `#import` directive generates to call the TestStand ActiveX API instead of using the Class Wizard tool to generate Microsoft Foundation Class (MFC) wrapper class files.

Select **Insert ActiveX Control** from the dialog box context menu and select a control that begins with `TestStand UI` to add a TestStand UI Control to a dialog box as a resource.

When you programmatically create a TestStand UI Control in an MFC container, you must remove the `WS_CLIPSIBLINGS` style from the control window for the TestStand UI Control to remain visible inside an MFC Group Box control. If you do not remove the `WS_CLIPSIBLINGS` style, a native MFC control always obscures the TestStand UI Control, even when the MFC control comes after the TestStand UI Control in the tab

order.

Obtaining an Interface Pointer and CWnd for an ActiveX Control

Complete the following steps to obtain an interface pointer to an ActiveX control, such as a TestStand User Interface (UI) control, that you insert into an Microsoft Foundation Class (MFC) dialog resource.

1. Add a CWnd member to the dialog class for the control as follows:

```
CWnd
mExprEditCWnd;
```
2. Insert the following code into the OnInitDialog method of the dialog class:

```
mExprEditCWnd.Attach(GetDlgItem
(IDC_MYEXPRESSIONEDIT)->m_hWnd);
```
3. Obtain the interface pointer from the CWnd member as follows:

```
TSUI::IExpressionEditPtr myExprEdit =
mExprEditCWnd.GetControlUnknown();
```



Note NI does not recommend using `DoDataExchange` to obtain an interface pointer and CWnd for a TestStand ActiveX User Interface Control because the pointer can be invalid in some instances. Use `DoDataExchange` only when controls are windowless or do not recreate internal windows.

Related concepts:

- [TestStand UI Controls](#)
- [TestStand Utility Functions Library](#)
- [TestStand ActiveX API Overview](#)
- [Programming with the TestStand API in Microsoft Visual C++](#)

Getting Started with User Interface Development

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

Related concepts:

- [Manager Controls](#)
- [Visible Controls](#)
- [Creating Custom User Interfaces](#)

Manager Controls

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

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

Related concepts:

- [Application Manager](#)
- [SequenceFileView Manager](#)
- [ExecutionView Manager](#)
- [Connecting Manager Controls to Visible Controls](#)
- [TestStand UI Controls](#)

Application Manager

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

- Processes command-line arguments.
- Maintains an application configuration file.
- Initializes and shuts down the TestStand Engine.
- Loads type palette files at initialization.
- Logs users in and out.
- Loads and unloads sequence files.
- Launches executions.
- Tracks existing sequence files and executions.

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

SequenceFileView Manager

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

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

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

Related concepts:

- [TestStand UI Controls](#)

ExecutionView Manager

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

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

which the ExecutionView Manager control connects.

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

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

Related concepts:

- [TestStand UI Controls](#)

Visible Controls

The TestStand User Interface (UI) Controls in the following table are visible at design time and run time and are similar to common Microsoft Windows UI controls. Connect visible TestStand UI Controls to manager controls to display information or to allow users to select items to view.

Control Name	Description
Button	Connect a manager control to a Button control to specify that the button performs a common user interface command, such as "Open Sequence File." The Button control uses a localized caption and automatically enables or disables according to the application state.
CheckBox	Connect a manager control to a CheckBox control so users can toggle the state of a common user interface command, such as "Break on Step Failure."
ComboBox	Connect a manager control to a ComboBox control so users can view or select from a list of adapters, sequence files, sequences, step groups, executions, threads, or stack frames.
ExpressionEdit	Use an ExpressionEdit control so users can edit a TestStand expression with syntax coloring, popup help, and statement completion. Although you typically do not need to edit expressions in a user interface

Control Name	Description
	application, you can connect a manager control to a read-only <code>ExpressionEdit</code> control to display text information about the application state, such as the pathname of the selected sequence file or the name of the current user. You can also use <code>ExpressionEdit</code> controls in dialog boxes for step types and in tools in which you prompt users to enter a TestStand expression.
<code>InsertionPalette</code>	Connect a <code>SequenceFileView</code> Manager control to an <code>InsertionPalette</code> control so users can insert steps and template items into a sequence file by dragging or double-clicking.
<code>Label</code>	Connect a manager control to a <code>Label</code> control to display text information about the application state in the label, such as the name of the current user or the status of the current unit under test (UUT).
<code>ListBar</code>	Use a <code>ListBar</code> control to display multiple pages, where each page contains a list of items users can view or select. Connect a manager control to a <code>ListBar</code> page so users can view and select from a list of adapters, sequence files, sequences, step groups, executions, threads, or stack frames.
<code>ListBox</code>	Connect a manager control to a <code>ListBox</code> control so users can view or select from a list of adapters, sequence files, sequences, step groups, executions, threads, or stack frames.
<code>ReportView</code>	Connect an <code>ExecutionView</code> Manager control to a <code>ReportView</code> control to display the report for the selected execution.
<code>SequenceView</code>	Connect a <code>SequenceFileView</code> Manager control or an <code>ExecutionView</code> Manager control to a <code>SequenceView</code> control to display the steps of a sequence from a sequence file or execution. The <code>SequenceView</code> control displays the steps in a list with columns you specify when you configure the control.
<code>StatusBar</code>	Connect a manager control to panes of a <code>StatusBar</code> control to display textual, image, or progress information about the application state. You can programmatically control individual <code>StatusBar</code> panes to display custom information.
<code>VariablesView</code>	Connect a <code>SequenceFileView</code> Manager control or an <code>ExecutionView</code> Manager control to a <code>VariablesView</code> control to display the sequence context for the sequence file or execution.

Related concepts:

- [TestStand UI Controls](#)
- [Manager Controls](#)
- [SequenceFileView Manager](#)
- [ExecutionView Manager](#)

Connecting Manager Controls to Visible Controls

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

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

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

Related concepts:

- [Manager Controls](#)
- [Visible Controls](#)

View Connections

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

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

Connect a SequenceFileView Manager control or an ExecutionView Manager control to

a VariablesView control to display the sequence context for the sequence file or execution.

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

Call the following methods to connect to view controls:

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

Related concepts:

- [Manager Controls](#)
- [TestStand UI Controls](#)
- [SequenceFileView Manager](#)
- [ExecutionView Manager](#)
- [Sequence Context](#)
- [Visible Controls](#)

List Connections

You can connect a ComboBox control, a ListBox control, or a ListBar page to a list a manager control provides, as shown in the following table.

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

List	Manager Control
Threads	ExecutionView Manager
Stack frames	ExecutionView Manager

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

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

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

Related concepts:

- [Manager Controls](#)
- [Application Manager](#)
- [SequenceFileView Manager](#)
- [ExecutionView Manager](#)

Command Connections

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

common to most TestStand applications.

The `CommandKinds` enumeration in the TestStand User Interface (UI) Controls API defines a set of common commands you can add to an application. You can connect these commands to TestStand buttons or application menu items, which automatically execute the command. You do not need an event handler to implement the command.

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

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

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

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

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

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

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

Related concepts:

- [Manager Controls](#)
- [ExecutionView Manager](#)
- [Managing Menus and Menu Items](#)

Information Source Connections

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

Related concepts:

- [Manager Controls](#)
- [Caption Connections](#)
- [Image Connections](#)
- [Numeric Value Connections](#)

Caption Connections

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

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

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

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

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

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

Related concepts:

- [Application Manager](#)
- [Manager Controls](#)
- [ExecutionView Manager](#)

Image Connections

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

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

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

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

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

- `ApplicationMgr.GetImageName`

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

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

Related concepts:

- [ExecutionView Manager](#)
- [Manager Controls](#)
- [SequenceFileView Manager](#)

Numeric Value Connections

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

The `NumericSources` enumeration defines the set of values to which you can connect.

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

Related concepts:

- [ExecutionView Manager](#)

Specifying and Changing Control Connections

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

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

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

Related concepts:

- [Manager Controls](#)
- [Visible Controls](#)
- [SequenceFileView Manager](#)

Handling Events

TestStand User Interface (UI) Controls generate events to notify an application of user input and of application events, such as an execution completing. The visible controls generate user input events, such as `KeyDown` or `MouseDown`. The manager controls generate application state events, such as the `ApplicationMgr.SequenceFileOpened` event or the `ApplicationMgr.UserChanged` event. You can handle events according to the needs of the application, as shown in the following table.

ADE	Description
LabVIEW	Register event handler VIs with the Register Event Callback function.
LabWindows/ CVI	Install ActiveX event callback functions by calling the <code>TSUI_<object class>EventsRegOn<event name></code> functions in <code>tsui.fp</code> .
.NET	Create .NET control event handlers from the form designer.
C++ (MFC)	Create ActiveX event handlers from the Message Maps page in the Class Wizard dialog box.

Related concepts:

- [TestStand UI Controls](#)
- [Visible Controls](#)
- [Manager Controls](#)
- [Handling Events in a LabVIEW User Interface](#)
- [Handling Events in a LabWindows/CVI User Interface](#)

Events Typical Applications Handle

When you create an application, you can direct the application to handle any subset of the available TestStand User Interface (UI) Control events. However, an application typically handles the events described in the following table.

Event	Description
ExitApplication	The Application Manager control generates this event to request that the application exit. Handle this event by directing the application to exit normally.
Wait	The Application Manager control generates this event to request that the application display or remove a busy indicator. Handle this event by displaying or removing a wait cursor according to the value of the showWait event parameter.
ReportError	The Application Manager control generates this event to request that the user interface return an error during user input or during an asynchronous operation. Handle this event by displaying the error code and description in a dialog box or by appending the error code and description to an error log. The ApplicationMgr.ReportError event indicates an application error, not a sequence execution error. The ApplicationMgr.BreakOnRunTimeError event indicates a sequence execution error.
DisplaySequenceFile	The Application Manager control generates this event to request that the application display a particular sequence file. Handle this event by displaying the sequence file by setting the SequenceFileViewMgr.SequenceFile property. • If the application has only a single window, set this property on the SequenceFileView Manager control that resides on the window. • If the application displays each sequence file in a separate window using separate SequenceFileView Manager controls, call

Event	Description
	the ApplicationMgr.GetSequenceFileViewMgr method to find the SequenceFileView Manager control that currently displays the sequence file so you can activate the window that contains the sequence file. If no SequenceFileView Manager control currently displays the sequence file, a multiple window application can create a new window that contains a SequenceFileView Manager control. The application can then set the <code>SequenceFileViewMgr.SequenceFile</code> property to display the sequence file in the new window.
DisplayExecution	The Application Manager control generates this event to request that the application display a particular execution. Handle this event by displaying the execution by setting the ExecutionViewMgr.Execution property. - If the application has only a single window, set this property on the ExecutionView Manager control that resides on the window. - If the application displays each execution in a separate window using separate ExecutionView Manager controls, call the ApplicationMgr.GetExecutionViewMgr method to find the ExecutionView Manager control that currently displays the execution so you can activate the window that contains the execution. - If no ExecutionView Manager control currently displays the execution, a multiple window application can create a new window that contains an ExecutionView Manager control. The application can then set the <code>ExecutionViewMgr.Execution</code> property to display the execution in the new window.

Related concepts:

- [Application Manager](#)
- [Startup and Shutdown](#)
- [SequenceFileView Manager](#)
- [ExecutionView Manager](#)

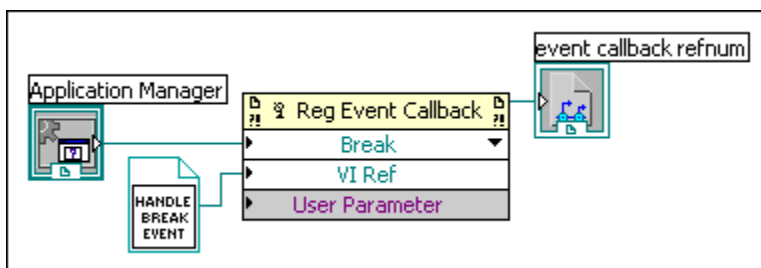
Handling Events in a LabVIEW User Interface

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

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

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

The following figure shows how to register a callback VI to handle the Break event for the Application Manager control:



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

Optimizing Event Handling Performance

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

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

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

Related concepts:

- [Handling Events](#)
- [Example User Interfaces](#)
- [TestStand UI Controls](#)

Handling Events in a LabWindows/CVI User Interface

To handle an event in LabWindows/CVI, you register a callback function, which LabWindows/CVI automatically calls when the control generates the event. Use the Event Callback Registration functions in the TestStand User Interface (UI) Controls driver to perform event registration.

For example, the following statement registers a callback function for the `OnExitApplication` event sent from the Application Manager control:

```
TSUI_ApplicationMgrEventsRegOnExitApplication
(gAppMgrHandle, AppMgr_OnExitApp, NULL, 1, NULL);
```

The callback function can contain the following code, which verifies whether the TestStand Engine is in a state where it can shut down:

```
HRESULT CVICALLBACK AppMgr_OnExitApp(CAObjHandle caServerObjHandle, void
*caCallbackData)
{
    VBOOL canExitNow;
    if (!TSUI_ApplicationMgrShutdown(gAppMgrHandle, &errorInfo, &canExitNow) &&
        (canExitNow))
        QuitUserInterface(0);
    return S_OK;
}
```

Related concepts:

- [Handling Events](#)
- [TestStand UI Controls](#)

Startup and Shutdown

As a final step in the initialization of the application, call the `ApplicationMgr.Start` method to initialize the Application Manager control and launch the LoginLogout Front-End callback sequence if you did not set the `ApplicationMgr.LoginOnStart` property to `False`.

Complete the following steps to shut down the application.

1. When the application holds any references to TestStand objects, such as sequence files or executions, handle the `ApplicationMgr.QueryShutdown` event by canceling the shutdown process or releasing the TestStand object references the application holds.
2. Call the `ApplicationMgr.Shutdown` method. When the method returns `True`, exit the application. When the method returns `False`, do not exit the application. Leaving the application running allows the method to shut down any running executions and unload sequence files. When the shut down process completes, TestStand User Interface (UI) Controls the Application Manager control generates the `ApplicationMgr.ExitApplication` event to notify you to exit the application. If the application cancels the shutdown process, the Application Manager control generates the `ApplicationMgr.ShutDownCancelled` event, which occurs when users choose not to terminate a busy execution.



Note When you use the TestStand User Interface (UI) Controls to create an **Exit** button or an Exit menu option that invokes the `Exit` command, the button or menu option automatically calls the `ApplicationMgr.Shutdown` method for you.

3. Exit the application in the event handler you create for the `ApplicationMgr.ExitApplication` event. The window in which the Application Manager control resides must exist until you receive the `ApplicationMgr.ExitApplication` event.

Related concepts:

- [Application Manager](#)
- [Modifying Front-End Callbacks](#)
- [TestStand UI Controls](#)

Starting and Shutting Down TestStand from a LabVIEW User Interface

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

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

changes.

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

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

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

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

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

Related concepts:

- [Example User Interfaces](#)

Starting and Shutting Down TestStand from a LabWindows/CVI User Interface

When you initialize the user interface application, use the `TSUI_ApplicationMgrStart` driver function to invoke the `ApplicationMgr.Start` method, which starts the TestStand Engine and logs in a user.

LabWindows/CVI applications typically wait for user input by calling the `RunUserInterface()` function after loading and displaying the main user interface panel. The `RunUserInterface()` function handles all events, such as menu selections, control value changes, and ActiveX control events.

Typically, you click the **Close** box or execute the Exit command through a TestStand menu or a Button control to stop a user interface application. For user interface events that request the user interface to close, the user interface must call the `TSUI_ApplicationMgrShutdown` function to unload sequence files, log out, and trigger an `OnApplicationCanExit` event. When the function determines that the engine can shut down, the **canExitNow** output parameter returns `True`. The user interface application then calls the `QuitUserInterface()` function, which causes the preceding `RunUserInterface()` call to return. After the application exits the function call to `RunUserInterface()`, the user interface application must call `TSUI_ApplicationMgrShutdown` a second time to complete the cleanup process and shut down the engine.

Related concepts:

- [Startup and Shutdown](#)
- [TestStand UI Controls](#)

Starting and Shutting Down TestStand from a .NET User Interface

When you initialize the user interface application, use the `ApplicationWrapper.Run` method in the `NationalInstruments.TestStand.Utility` namespace to load the main form for the application that contains the TestStand Application Manager and other TestStand UI controls. The `Run` method calls `System.Windows.Forms.Application.Run`, which handles all events, such as menu selections, control value changes, and ActiveX control events. The method also works around exceptions that can occur on shutdown.

Use the `MainForm_Load` event to initialize the TestStand UI controls and use the `ApplicationMgr.Start` method to initialize the TestStand engine and log in a user.

Typically, you click the Close box or execute the Exit command through a TestStand menu or button control to stop an application. When operations request the form to

close, the form raises the `Closing` event, where you must call the `ApplicationMgr.Shutdown` method to initiate shutdown of TestStand. If the return value from the `Shutdown` method indicates that shutdown is not complete, you must cancel the `Close` event. When shutdown is complete, TestStand raises the `ApplicationMgr.ExitApplication` event, where you call the `Close` method on the form.

Refer to the simple user interface examples for C# and .NET to see how to properly start and shut down TestStand.

Additional Considerations for .NET Applications to Properly Shut Down TestStand

The TestStand engine shutdown process requires the proper release of all references to TestStand objects, such as variables, sequence files, and the TestStand engine itself. When using the TestStand engine in a .NET application, you must consider the following challenges:

- **TestStand reports object leaks**—The .NET CLR uses a form of automatic memory management, which includes garbage collection to reclaim memory, that is no longer in use. When the last reference to a TestStand COM object is released, .NET places the object reference into garbage collection, and the object is finalized at a later time. If the TestStand engine performs a shutdown before garbage collection finalizes TestStand objects, and you have the TestStand debug option **Report Object Leaks** enabled, TestStand reports non-finalized objects as leaks, even though the application no longer holds a reference to the objects.
- **TestStand engine cannot shut down**—Some global objects of .NET can retain references to TestStand objects that are passed to events of TestStand controls on Windows forms. If the application exits before releasing these references, the TestStand engine cannot perform a shutdown and some necessary tasks, such as saving some TestStand files.

To work around these challenges, NI recommends designing a TestStand .NET application to use a separate application domain instead of the default application domain and to force garbage collection and disposal of the TestStand engine.

A TestStand .NET application should perform the following startup and shutdown tasks:

1. Create the TestStand engine in the default application domain.
2. Create a new application domain and perform the following operations in that domain:
 - Create the TestStand engine to obtain a reference to the existing engine created by the default application domain.
 - Perform any startup operations on the engine object.
 - Perform operations that use the engine object, including loading and running the main form for the application.
 - Close any forms loaded into the application domain.
 - Release all TestStand objects, including the engine, created in the application domain.
 - Unload the application domain to ensure that all TestStand object references are finalized.
3. Force garbage collection of all object generations.
4. Force the release and finalization of the TestStand Engine COM object created in the default application domain.

To perform many of the previously listed tasks, you can call the `LaunchTestStandApplicationInNewDomain.LaunchProtected` method in the `NationalInstruments.TestStand.Utility` assembly. The `LaunchProtected` method creates an instance of the TestStand engine, loads a new application domain, calls a delegate main entry point in the application domain, and, before returning, the method properly unloads the application domain, forces garbage collection, and finalizes the TestStand engine COM object.

To properly cleanup before it returns, the `LaunchTestStandApplicationInNewDomain.LaunchProtected` method uses `TSHelper.DoSynchronousGCForCOMObjectDestruction` to force garbage collection by calling `System.GC.Collect` and `GC.WaitForPendingFinalizers` twice and calling `System.Runtime.InteropServices.Marshal.FinalReleaseComObject` on the engine reference.



Note Calling `FinalReleaseComObject` does not cause the underlying Runtime Callable Wrapper (RCW) object to be garbage collected, and any subsequent method calls on the RCW will cause an

InvalidComObjectException to be thrown.



Note You must use the `TestStandObjectReleaser` utility class in the `NationalInstruments.TestStand.Utility` DLL to dispose of `TestStand` objects in `TestStand 2025 Q2` and later, as `LaunchTestStandApplicationInNewDomain.LaunchProtected` does not create a new App domain in these `TestStand` versions.

Refer to the source code file, `<TestStand>\API\DotNET\Source\TSUtil\TSUtil.cs`, for more information about the implementation of the `LaunchTestStandApplicationInNewDomain` class and the `TSHelper` class that it uses.

The following example code illustrates the recommended way to use the `TestStand` engine in .NET:

```
using NationalInstruments.TestStand.Utility;
using NationalInstruments.TestStand.Interop.API;
[STAThread]
static void Main(string[] args)
{
    // Create application domain, call MainEntryPoint, and
    // cleanup before return.
    LaunchTestStandApplicationInNewDomain.LaunchProtected(
        new LaunchTestStandApplicationInNewDomain.
            MainEntryPointDelegateWithArgs(MainEntryPoint),
        args,
        "TestStand Application",
        new LaunchTestStandApplicationInNewDomain.
            DisplayErrorMessageDelegate(DisplayErrorMessage));
}
// Main entry point executed in new application domain.
private static void MainEntryPoint(string[] args)
{
    // Obtain a reference to existing Engine.
    Program._engine = new Engine();
    Program._engine.LoadTypePaletteFilesEx(
        TypeConflictHandlerTypes.ConflictHandler_Prompt, 0);
    // Launch your application here.
    // e.g., Application.Run(new MainForm());
}
```

```

    // Engine clean up.
    Program._engine = null;
}
// Called if exception occurs in MainEntryPoint.
private static void DisplayErrorMessage(string caption, string message)
{
    // Handle error here.
    // e.g., MessageBox.Show(message, caption);
}
private static Engine _engine = null;

```

Refer to the simple user interface examples for C# and .NET to see how to use a separate application domain to properly shut down TestStand.

Related concepts:

- [TestStand UI Controls](#)
- [Managed Code Considerations for TestStand and .NET 8](#)

License Checking

The `ApplicationMgr.Start` method verifies that a license exists to run the application. When no license exists, the `ApplicationMgr.Start` method returns an error the application displays before exiting. When a license that is not activated or an evaluation license exists, the `ApplicationMgr.Start` method prompts users to activate a license.

When the `ApplicationMgr.IsEditor` property is `True`, the `ApplicationMgr.Start` method requires a license that permits editing. When you call the `ApplicationMgr.Start` method when the `ApplicationMgr.IsEditor` property is `False` and later set the `ApplicationMgr.IsEditor` property to `True`, the `ApplicationMgr.IsEditor` property returns an error when it cannot obtain a license that permits editing.



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

Editor Versus Operator Interface Applications

An Editor application permits users to create, edit, and save sequence files. An Operator Interface application permits users only to run sequences.

Use the TestStand User Interface (UI) Controls to create Editor applications, Operator Interface applications, and applications that can switch between Editor Mode and Operator Mode.

Related concepts:

- [TestStand UI Controls](#)

Creating Editor Applications

You must enable the Editor Mode for the TestStand User Interface (UI) Controls to create an Editor application.

Set the `ApplicationMgr.IsEditor` property at edit time to specify whether Editor Mode is on or off by default. Alternatively, you can set the `ApplicationMgr.IsEditor` property in the application source code before you call the `ApplicationMgr.Start` method.

You can pass a command-line argument to override the default editing mode for the application. Pass `/editor` to set the `ApplicationMgr.IsEditor` property and pass `/operatorInterface` to clear the `ApplicationMgr.IsEditor` property. Set the `ApplicationMgr.CommandLineCanChangeEditMode` property to `False` to prevent users from changing the `ApplicationMgr.IsEditor` property from the command line.

The full-featured user interface examples allow users with sequence file editing permissions to toggle the editing mode by pressing <Ctrl-Alt-Shift-Insert>. To change or disable this keystroke in an application based on a full-featured example, set the `ApplicationMgr.EditModeShortcutKey` and `ApplicationMgr.EditModeShortcutModifier` properties in the designer or in the user interface source code.

Related concepts:

- [TestStand UI Controls](#)

User Interface Development Best Practices

Learn the best practices for developing user interfaces.

Localizing User Interfaces

The `StationOptions.Language` property specifies the current language. Localized TestStand applications use the `Engine.GetResourceString` method to obtain text in the current system language from language resource files.

Call the `ApplicationMgr.LocalizeAllControls` method to localize all the user-visible TestStand User Interface (UI) Control strings you configure at design time. Using the `ApplicationMgr.LocalizeAllControls` method reduces the number of strings you must explicitly localize using the `Engine.GetResourceString` method by localizing items such as list column headers in the `SequenceView` control, text in the `StatusBar` pane, captions in the `Button` control, and captions in the `ListBar` page.

Buttons and menu items you connect to commands automatically localize caption text.

The `ApplicationMgr.LocalizeAllControls` method operates only on TestStand UI Controls. For other controls and user interface elements, the application must set each item of localized text. The following table lists the TestStand Utility (TSUtil) library functions you can use to localize non-TestStand controls and menu items.

ADE	TSUtil Library Localization Function
LabVIEW	<code>TestStand - Get Resource String.vi</code> <code>TestStand - Localize Front Panel.vi</code> <code>TestStand - Localize Menu.vi</code> Refer to the Full UI - Localize Operator Interface and to the Full UI - About Box example user interface VIs for examples of localizing user interfaces.
LabWindows/ CVI	<code>TS_LoadPanelResourceStrings</code> <code>TS_LoadMenuBarResourceStrings</code> <code>TS_SetAttrFromResourceString</code> <code>TS_GetResourceString</code> <code>TSUI_ApplicationMgrLocalizeAllControls</code> Refer to the <TestStand

ADE	TSUtil Library Localization Function
	Public>\UserInterfaces\Full-Featured\CVI\Source Code\TestExec.c source file for an example of localizing user interface panels.
.NET	Localizer.LocalizeForm Localizer.LocalizeMenu
C++ (MFC)	Localizer.LocalizeWindow Localizer.LocalizeMenu Localizer.LocalizeString

Related concepts:

- [Creating String Resource Files](#)
- [TestStand UI Controls](#)
- [Command Connections](#)
- [TestStand Utility Functions Library](#)
- [Example User Interfaces](#)
- [TestStand Directory Structure](#)

Documenting Custom User Interfaces

You can use the Using the NI TestStand User Interfaces document, located at <TestStand>\Doc\UsingtheTestStandUserInterfaces.html, as a starting point for creating custom documentation for user interface applications you customize based on the TestStand full-featured user interface examples.

In addition, the menus for the TestStand full-featured user interface examples include the CommandKind_DefaultHelpMenu_Set constant, which contains a set of commands that corresponds to the typical items in the Help menu of a TestStand application, including support for using the <F1> key to display help for the currently active TestStand User Interface control.

Related concepts:

- [TestStand Directory Structure](#)
- [TestStand UI Controls](#)

User Interface Application Styles

Although you can use the TestStand User Interface (UI) Controls to create any type of application, the following formats are the most common:

- Single window
- Multiple window
- No visible window

Applications of a particular style usually share a similar implementation strategy, particularly with respect to the use of the TestStand manager controls.

Related concepts:

- [TestStand UI Controls](#)
- [Single Window Applications](#)
- [Multiple Window Applications](#)
- [No Visible Window Applications](#)
- [Manager Controls](#)

Single Window Applications

A single window application typically displays one execution and sequence file at a time. Users can select the execution and sequence file to display from a ListBar, ComboBox, or ListBox control. The examples in the <TestStand>\UserInterfaces\Full-Featured and <TestStand>\UserInterfaces\Simple directories are single window applications.

A single window application contains one Application Manager control, one SequenceFileView Manager control, and one ExecutionView Manager control. To display sequences, connect the SequenceFileView Manager and ExecutionView Manager controls to separate SequenceView controls, alternate a connection from each manager control to a single SequenceView control, or leave one or both manager controls unconnected to a SequenceView control.

In the examples in the Full-Featured directory, the SequenceFileView Manager control and the ExecutionView Manager control connect to separate SequenceView controls, and only one SequenceView control is visible at a time. Visibility depends

on whether you select to view sequence files or executions.

In the examples in the `Simple` directory, the `ExecutionView` Manager control connects to the `SequenceView` control. Because the `SequenceFileView` Manager control does not connect to a `SequenceView` control, these examples display only sequences for the current execution, not sequences from the sequence file selection.

Related concepts:

- [TestStand Directory Structure](#)
- [Application Manager](#)
- [SequenceFileView Manager](#)
- [ExecutionView Manager](#)

Multiple Window Applications

A multiple window application includes at least one window that always exists to contain the Application Manager control. Although this window can be visible or invisible, it is typically visible and contains controls for users to open sequence files.

Sequence File Window

For each sequence file users open, the application creates a Sequence File window that contains a `SequenceFileView` Manager control and a `SequenceView` control to which the manager control connects. The application sets the `SequenceFileViewMgr.UserData` property to attach a handle, reference, or pointer that represents the window. When the application receives the `ApplicationMgr.DisplaySequenceFile` event, the application calls `ApplicationMgr.GetSequenceFileViewMgr` to determine whether a `SequenceFileView` Manager control currently displays the sequence file. If so, the application retrieves the window from the `SequenceFileViewMgr.UserData` property and activates the window. If no window currently displays the sequence file, the application creates a new window and sets the `SequenceFileViewMgr.SequenceFile` property to display the sequence file. Because the window displays only this sequence file, the application also sets the `SequenceFileViewMgr.ReplaceSequenceFileOnClose` property to `False`.

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

closes the window immediately. When the `SequenceFileViewMgr.SequenceFile` property is not `NULL`, the application does not close the window. Instead, the application passes the sequence file to the `ApplicationMgr.CloseSequenceFile` method. When the application receives the `SequenceFileViewMgr.SequenceFileChanged` event with a `NULL` sequence file event parameter, the application closes the window that holds the `SequenceFileView` Manager control.

Execution Window

The Sequence File window contains controls for users to execute the sequence file the window displays. For each execution users start, the application creates an Execution window that contains an `ExecutionView` Manager control and a `SequenceView` control to which the manager control connects. The application sets the `ExecutionViewMgr.UserData` property to attach a handle, reference, or pointer that represents the window. When the application receives the `ApplicationMgr.DisplayExecution` event, the application calls the `ApplicationMgr.GetExecutionViewMgr` method to determine whether an `ExecutionView` Manager control currently displays the execution. If so, the application retrieves the window from the `ExecutionViewMgr.UserData` property and activates the window. If no window currently displays the execution, the application creates a new window and sets the `ExecutionViewMgr.Execution` property to display the execution. Because the window displays only this execution, the application also sets the `ExecutionViewMgr.ReplaceExecutionOnClose` property to `False`.

When an Execution window attempts to close and the `ExecutionViewMgr.Execution` property is `NULL`, the application closes the window immediately. When the `ExecutionViewMgr.Execution` property is not `NULL`, the application does not close the window. Instead, the application passes the execution to the `ApplicationMgr.CloseExecution` method. The application does not immediately close the Execution window to ensure that the window exists until the execution the window displays completes. When the application receives the `ExecutionViewMgr.ExecutionChanged` event with a `NULL` execution event parameter, the application closes the window that holds the `ExecutionView` Manager control.

Child Windows

A multiple window application can display multiple child windows instead of displaying sequence files and executions in separate top-level windows. Child windows can be visible or reside on tab control pages or similar locations that allow users to easily select which child window to view.

Related concepts:

- [Application Manager](#)
- [SequenceFileView Manager](#)
- [ExecutionView Manager](#)

No Visible Window Applications

An application without a visible window is similar to a single window application. The application can execute command-line arguments and then exit, or the application can have a different mechanism to determine which files to load and execute.

Although an invisible application does not require an ExecutionView Manager control, the application can use a SequenceFileView Manager control to provide methods to launch an execution for a sequence file. Use the SequenceFileViewMgr.ExecutionEntryPoints property, the SequenceFileViewMgr.Run method, the SequenceFileViewMgr.RunSelectedSteps method, the SequenceFileViewMgr.LoopOnSelectedSteps method, and the SequenceFileViewMgr.GetCommand method to launch executions in an application without a visible window.

Related concepts:

- [Single Window Applications](#)
- [ExecutionView Manager](#)
- [SequenceFileView Manager](#)

Persisting Application Settings

The TestStand Engine stores Station Options dialog box settings, Insertion Palette step type and template list settings, and other settings that apply to all TestStand

applications in files such as `GeneralEngine.cfg` and `Templates.ini`, located in the `<TestStand Application Data>\Cfg` directory. However, each user interface also stores additional custom settings, including breaking on the first step of execution, breaking when a step fails, and listing the most recently used sequence files. The Application Manager control stores these settings in the configuration file the `ApplicationMgr.ConfigFilePath` property specifies.

The `ApplicationMgr.BreakOnFirstStep`, `ApplicationMgr.PromptForOverwrite`, `ApplicationMgr.EditReadOnlyFiles`, `ApplicationMgr.MakeStepNamesUnique`, and `ApplicationMgr.SaveOnClose` properties persist to the configuration file. Setting the value of one of these properties on the Application Manager control in a designer sets the default value for the property. The Application Manager control stores the default value in the configuration file it creates when a configuration file does not already exist. When the configuration file already exists, the Application Manager control loads the values of these properties from the file.

Related concepts:

- [TestStand Directory Structure](#)
- [Application Manager](#)

Specifying the Configuration File Location

The default value of the `ApplicationMgr.ConfigFilePath` property is `%TestStandLocalAppData%\UserInterface.xml`, in which `%TestStandLocalAppData%` is a macro that expands to a directory to which the currently logged-in user has permission to write files. The directory is typically `<User Directory>\AppData\Local\National Instruments\TestStand on Microsoft Windows 10/8.1/7`. Set the `ApplicationMgr.ConfigFilePath` property before the application calls the `ApplicationMgr.Start` method to change the configuration file location.

When you specify a relative file path or just a filename, the file location is relative to the directory that contains the application. When users who do not have Windows administrator privileges can run the application, you must store the configuration file in a location to which users have permission to write files.

Related concepts:

- [TestStand Directory Structure](#)

Adding Custom Application Settings

After the application calls the `ApplicationMgr.Start` method, complete the following steps to add your own setting to persist in the configuration file.

1. Access the `ApplicationMgr.ConfigFile` property to obtain the `PropertyObjectFile` that holds the contents of the configuration file.
2. Access the `PropertyObjectFile.Data` property to obtain the `PropertyObject` that holds the application settings.
3. Ensure the custom setting exists in the `PropertyObject` by setting a default value of the setting by calling a method, such as the `PropertyObject.SetValBoolean` method, with a lookup string, such as `"CustomSettings.MyExampleBooleanSetting"`, and an **options** parameter of the `PropOption_SetOnlyIfDoesNotExist` constant.
4. Call a method, such as the `PropertyObject.GetValBoolean` method, to obtain the current value of the custom option.
5. Call a method, such as the `PropertyObject.SetValBoolean` method, with an **options** parameter of the `PropOption_NoOptions` constant to set the custom option in response to user input.

When you call the `ApplicationMgr.Shutdown` method or change any Application Manager control setting, the Application Manager control persists the application settings to the configuration file. You can also call the `PropertyObjectFile.WriteFile` method at any time to persist the settings.

Related concepts:

- [Application Manager](#)

Deploying a User Interface

You can use the TestStand Deployment Utility to deploy a user interface. Refer to the [Distributing a User Interface](#) and [Including a User Interface in a Deployment](#) topics for more information about deploying user interfaces.

Related concepts:

- [Including a User Interface in a Deployment](#)
- [Distributing a User Interface](#)
- [Deploying TestStand Systems](#)

Advanced User Interface Development

Learn how to develop advanced user interfaces.

TestStand Utility Functions Library

Use the TestStand Utility (TSUtil) Functions Library to use certain aspects of the TestStand API in particular application development environments (ADEs). Many TestStand Utility (TSUtil) functions operate on environment-specific objects, such as menus, that the environment-neutral TestStand API cannot access. The functions available in TSUtil vary according to the ADE.

The TSUtil library contains functions to insert menu options that automatically execute commands the TestStand User Interface (UI) Controls API provides. The TSUtil library also provides functions to help localize the strings on a user interface.

The following table describes how to use the TSUtil library in different ADEs. If the table does not include the ADE you use, a version of TSUtil does not exist for the ADE.

Table 41. Using the TSUtil Library in Different ADEs

ADE	Help Location	Files	How to Use
LabVIEW	Context help for each VI and in the NI TestStand VIs and Functions Help, accessible by right-clicking the VI and selecting Help from the shortcut menu or by selecting	VIs on the Functions»TestStand palette _TSUtility.llb located in <TestStand>\API\LabVIEW	Place VIs on the block diagram

ADE	Help Location	Files	How to Use
	Help»NI TestStand VIs and Functions		
LabWindows/ CVI	Function panels (TSUtil.fp)	TSUtil.c, TSUtil.h, TSUtil.fp, and TSUtil.obj located in <TestStand>\API\CVI	Insert TSUtil.fp into the LabWindows/ CVI project. Include TSUtil.h and TSUtil.c source files as needed. The n- TestStand-related functions begin with the TS_ prefix.
.NET	In the Object Browser and in the source window using Microsoft IntelliSense	National Instruments.TestStand.Utility.dll located in <TestStand>\API\DotNet\ Assemblies\CurrentVersion	Add a reference to the assembly in the project. The classes in this assembly reside in the National Instruments.TestStand namespace.
C++ (MFC)	Comments in the C++ header file, TSUtilCPP.h	TSUtilCPP.cpp and TSUtilCPP.h located in <TestStand>\API\VC	Add TSUtilCPP.cpp to the project. Include TSUtilCPP.h once. Include TSUtilCPP.h source files as needed. The class library reside in the TSUtil

You can use the source code for one of the existing TSUtil libraries as a guide to write your own code that performs similar functionality.

Related concepts:

- [TestStand ActiveX API Overview](#)
- [Managing Menus and Menu Items](#)
- [TestStand UI Controls](#)
- [Localizing User Interfaces](#)
- [TestStand Directory Structure](#)
- [Adding Assembly References in Microsoft Visual Studio](#)
- [Using TestStand UI Controls in LabVIEW](#)
- [Using TestStand UI Controls in LabWindows/CVI](#)

Adding Assembly References in Microsoft Visual Studio

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

1. Select the project in the Solution Explorer.
2. Select **Project»Add Reference** to launch the Add Reference dialog box.
3. (Visual Studio 2010 and earlier) Click the **.NET** tab and select **TestStand <Version> Utilities** from the list of components. (Visual Studio 2012 and later) Click the **Assemblies** item and select **Extensions**. Select **TestStand <Version> Utilities** from the list of components.
4. Click **OK** to close the Add Reference dialog box.



Note

- TestStand <Version> Utilities is the name for the `NationalInstruments.TestStand.Utility` assembly.
- In Visual Studio 2012 and later, the Add Reference dialog box is named the Reference Manager dialog box.

When you add an assembly reference, you must also complete the following steps to change the value of the Embed Interop Types property of the referenced assembly to `False`.

1. In the Solution Explorer, select the referenced assembly in the References section of the project.
2. In the Properties window, change the value of the Embed Interop Types property to `False`.

Managing Menus and Menu Items

TestStand applications that provide non-trivial menus can require a large amount of source code to build and update the state of menus and to handle events for menu items. Use the TestStand Utility (TSUtil) functions to create menu items that invoke TestStand commands to greatly reduce the amount of code required to implement menus in an application. TestStand automatically dims or enables these menu items according to the application state and sets their captions according to the language selection. The menu items execute commands automatically so that the application does not need to handle menu events or provide command implementations.

The application can also insert sets of dynamic menu items, such as a set of menu

items to open files from the most recently used file list or a set of menu items that run the current sequence with each available process model entry point. To create TestStand menu items, you must first add TSUtil to the project.



Note In TestStand 2025 Q2, the TSUtil .NET menu functions support using the MenuStrip, ToolStripMenuItem, ToolStripSeparator, and ToolStripItem controls but do not support using the MainMenu control in Microsoft Visual Studio.



Note In TestStand 2024 Q2 and earlier, the TSUtil .NET menu functions support using the MainMenu control but do not support using the MenuStrip control in Microsoft Visual Studio. To access the .NET MainMenu control in the Visual Studio Toolbox, select **Choose Items** from the context menu on the Toolbox pane, enable **MainMenu 2.0** on the .NET Framework Components tab of the Choose Toolbox Items dialog box, and click **OK** to close the dialog box. The full-featured .NET example user interface applications use MainMenu to display menus.

Updating Menus and Menu Items

The contents of a menu can vary depending on the current selection, other user input, or asynchronous execution state changes. Instead of updating a menu in response to any event or change that might affect the menu, update the state of a menu just before the menu displays when the user opens the menu. The following table lists the notification method different application development environments (ADEs) use to notify the application when a user is about to open a menu.

ADE	Menu Open Notification Method
LabVIEW	<This VI>:Menu Activation? event
LabWindows/CVI	InstallMenuDimmerCallback
.NET	Form.MenuStart
C++ (MFC)	CWnd::OnInitMenuPopup

Use the `RemoveMenuCommands`, `InsertCommandsInMenu`, and `CleanupMenu`

TestStand Utility (TSUtil) functions in LabVIEW and LabWindows/CVI to handle the menu open notifications and remove and reinsert TestStand menu items. You can remove and insert TestStand commands in menus that contain non-TestStand menu items.

The `InsertCommandsInMenu` function accepts an array of `CommandKinds` enumeration constants. Depending on the element value and the application state, each array element can create a single menu item, a set of several menu items, or no menu items. The `CommandKinds` enumeration also provides constants that expand into the full set of items commonly found in test application top-level menus, such as the File menu, Debug menu, or Configure menu.

Refer to the examples in the `<TestStand>\UserInterfaces\Full-Featured` directory for sample code that handles menu open notification events.

Related concepts:

- [TestStand Utility Functions Library](#)
- [Managing Menu Bars and Menu Event Handling in a LabVIEW User Interface](#)
- [Managing Menu Bars in a LabWindows/CVI User Interface](#)
- [TestStand Directory Structure](#)

Managing Menu Bars and Menu Event Handling in a LabVIEW User Interface

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

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

Because maintaining the current state of the menu bar can be difficult, NI recommends that you handle the menu bar only when required. The Event structure in a main event loop can include a case to handle the Menu Activation? event to determine when you open a menu or select a shortcut key that might be linked to a menu item. The block diagram that handles this event can then rebuild the menu bar.

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

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

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

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

Related concepts:

- [Using TestStand UI Controls in LabVIEW](#)
- [Example User Interfaces](#)
- [TestStand UI Controls](#)

Managing Menu Bars in a LabWindows/CVI User Interface

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

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

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

Call the `TS_InsertCommandsInMenu` function when the application rebuilds the menu bar in a `MenuDimmerCallback` function to populate the menu bar with commands that apply to the current state of the application. Before you call this function, you can call `TS_RemoveMenuCommands` to remove any menu items you previously inserted.

Refer to the `RebuildMenuBar` function in the `<TestStand Public>\UserInterfaces\Full-Featured\CVI\Source Code\TestExec.c` source file for an example of rebuilding the menu bar.

Related concepts:

- [TestStand Directory Structure](#)
- [TestStand UI Controls](#)

Implementing Command-Line Arguments for a User Interface

The Application Manager control automatically processes the command-line argument that invokes the application when you call the `ApplicationMgr.Start` method. Set the `ApplicationMgr.ProcessCommandLine` property to `False` before you call the `ApplicationMgr.Start` method to disable command-line processing.

You can also handle the `ApplicationMgr.ProcessUserCommandLineArguments` event to support additional command-line arguments. The `ApplicationMgr.ProcessUserCommandLineArguments` event occurs when the Application Manager control parses and processes an unrecognized command-line flag.

Related concepts:

- [Application Manager](#)
- [Configuring Sequence Editor and User Interface Startup Options](#)

Authenticode Signatures for Microsoft Windows

Authenticode signatures can help identify the publisher of a binary file and can help ensure that a binary file has not been modified since publication. In addition, when an application launches on Microsoft Windows 10/8.1/7, User Account Control (UAC) determines whether to grant the application administrator privileges and displays your company as the publisher when confirming elevation. Refer to Microsoft documentation for more information about Authenticode signatures.

Add an Authenticode signature to a TestStand user interface you create when you plan to allow users to download the user interface from a non-trusted public site and you want the operating system to identify your company as the publisher of the user interface. Also add an Authenticode signature to a user interface you create when the user interface requires administrator privileges to run on Windows and you want the UAC elevation prompt to identify your company as the publisher of the user interface.

To verify an Authenticode signature, the requesting computer must connect to the Internet to obtain a current Certificate Revocation List (CRL). For .NET applications, the .NET Common Language Runtime (CLR) verifies Authenticode signatures for

assemblies. If the computer that loads the assembly is not connected to the Internet, the CLR waits 15 seconds before timing out.

Complete the following steps to disable CRL validation in Microsoft Internet Explorer to avoid the timeout period on the computer, even when the default browser on the computer is not Internet Explorer. Using the Internet Explorer Internet Options to disable CRL validation does not expose the computer to any additional security threats.

1. Navigate to the standard Windows Control Panel facility for Internet options and click the **Advanced** tab.
2. In the Security section, disable the **Check for publisher's certificate revocation** option.

Alternatively, you can disable CRL validation by setting the registry key value of `HKCU\Software\Microsoft\Windows\CurrentVersion\WinTrust\Trust Providers\Software Publishing\State` to `0x00023e00`. To enable CRL validation, set the registry key value to `0x00023c00`.

When you disable CRL validation to avoid the timeout period, the CLR does not validate Authenticode-signed assemblies and does not grant the assemblies publisher evidence or publisher identity permissions, which is the same result when a timeout occurs. If the assemblies need these permissions, the computer must connect to the Internet or you must download a current CRL every 10–15 days.

As an alternative to disabling CRL validation for the entire computer, you can work around CRL validation if an application that uses the .NET Framework 2.0 and that has an Authenticode signature experiences the 15-second load time delay. Microsoft provides a fix you can download so you can correct this delay for .NET Framework 2.0 applications. The .NET Framework 2.0 Service Pack 1 also includes this fix. Refer to the Microsoft support article at support.microsoft.com/kb/936707 for more information about correcting delays in .NET Framework 2.0 applications that use Authenticode signatures.

The TestStand Sequence Editor and user interface examples do not include Authenticode signatures because NI distributes TestStand through trusted channels and because the sequence editor and user interface examples do not require administrator privileges to run on Windows. Additionally, NI finds the 15-second load

time delay on isolated networks unacceptable and believes that you should use discretion when disabling CRL validation. Therefore, when you run the sequence editor or example user interfaces as administrator on Windows, the UAC elevation prompt does not identify the sequence editor or example user interface as a NI product.

Application Manifests

When an application launches on Microsoft Windows 10/8.1/7, the User Account Control (UAC) security component determines whether to grant the application administrative privileges. A user that logs into Windows as a standard user can write only to specific locations on disk and in the registry. Standard user is the default login.

Microsoft recommends that applications run without requiring administrator privileges. If you design applications that do not attempt to access protected areas of the operating system, all users can run the application as intended without requiring administrator privileges. You can also include manifests to specify the execution level the application requires.

When an application does not specify an execution level in a manifest, the UAC launches the application with the standard or administrator privileges of the user. With standard privileges, the system uses virtualization to redirect any read and write operations for system files and registry keys to a per-user location instead of the actual system copy of the file or registry key. Do not create applications that rely on virtualization to perform these types of administrative operations.

The default TestStand User Interface application binary files include manifests that instruct the UAC to execute the application without virtualization and without requiring administrative privileges. With LabVIEW, you can specify a custom manifest for built applications. With LabWindows/CVI, you can specify a manifest for built applications. When you build the application, refer to the documentation for the application development environment you used for more information about how to include a manifest.

Making Dialog Boxes Modal to TestStand

Code modules that TestStand calls can launch dialog boxes modal to TestStand

application windows, such as the TestStand Sequence Editor or custom user interfaces.

LabVIEW

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

LabWindows/CVI

Use the `TS_StartModalDialogEx` and `TS_EndModalDialog` functions the TestStand Utility Functions driver provides to make a dialog box modal to TestStand application windows. Refer to the `<TestStand>\Components\StepTypes\MessageBox\msgbox.c` source file for an example of to use these functions.

Related concepts:

- [TestStand Directory Structure](#)
- [TestStand Utility Functions Library](#)
- [Launching a Modal Dialog](#)

Migrating User Interfaces to .NET Core

Consider the following when migrating custom user interfaces to .NET Core:

1. Ensure that your custom UI applications work correctly on high-DPI monitors by making them DPI-unaware. Follow the below methods to disable DPI awareness for your UI in different programming languages:
 - For custom UIs developed in C#, make your application DPI-unaware by calling the `Application.SetHighDpiMode` method with the `HighDpiMode.DpiUnaware` parameter before running the application in your main method, like in the following example:

```
[STAThread]
```

```
private static void Main(string[] args)

{
    // Set the application to be DPI-unaware

    Application.SetHighDpiMode(HighDpiMode.DpiUnaware);

    // Your application initialization code here

}
```

- For custom UIs created with other programming languages, you can make them DPI-unaware by setting the `dpiAware` flag to `False` in the application's manifest file.

```
<dpiAware xmlns="http://schemas.microsoft.com/SMI/2005/
WindowsSettings">false</dpiAware>
```

2. Update the `.csproj` file or the runtime configuration file for your custom user interface to depend on `Microsoft.NETCore.App`, `Microsoft.WindowsDesktop.App`, and `Microsoft.AspNetCore.App`. You can use the below format to include dependencies on these .NET runtimes.

```
<ItemGroup>
<FrameworkReference
    Include="Microsoft.WindowsDesktop.App" />
<FrameworkReference
    Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

3. If you want to run LabVIEW with .NET in your custom UI, create a token file named `<ApplicationName>.ini` next to your application which includes the following content:

```
[LVRT]

DisableDotNetFrameworkSupport=True
```



Note This file must be created for any TestStand application that uses LabVIEW.

4. For .NET applications, set the `EnableUnsafeBinaryFormatterSerialization` property to `true` in the

.csproj file.

```
<PropertyGroup>

<EnableUnsafeBinaryFormatterSerialization>true</EnableUnsafeBinaryFormatterSer
ialization>
</PropertyGroup>
```

Related concepts:

- [Calling LabVIEW VIs That Invoke .NET Code in TestStand](#)

Example User Interfaces

The <TestStand>\UserInterfaces directory includes the following subdirectories that contain the executable, project, and source code files for each example user interface:

- Full-Featured—Contains user interfaces for loading, viewing, editing, saving, executing, and debugging sequence files. The full-featured examples include menus and localization options.
- Simple—contains similar but limited user interfaces with fewer commands and options but no menus. Also, the simple example user interfaces display the steps for executions you run but do not display steps for sequences you load.



Note TestStand no longer includes example user interfaces that do not use the TestStand UI Controls. These examples contained a large amount of complex source code, and they provided less functionality than the simpler examples that use the TestStand UI Controls. NI recommends using the examples that use the TestStand UI Controls as a basis for new development.

The following table lists all the simple and full-featured example user interfaces that TestStand provides:

ADE	Full-Featured User Interface	Simple User Interface
C++ (MFC)	<TestStand Public>\UserInterfaces\Full- Featured\C++ (MFC)\Source	<TestStand Public>\UserInterfaces\ Simple\C++ (MFC)\Source

ADE	Full-Featured User Interface	Simple User Interface
	Code\TestExec.vcproj	Code\TestExec.vcproj
C#	<TestStand Public>\UserInterfaces\Full-Featured\CSharp\Source Code\TestExec.VS2010.csproj	<TestStand Public>\UserInterfaces\Simple\CSharp\Source Code\TestExec.VS2010.csproj
LabVIEW	<TestStand Public>\UserInterfaces\Full-Featured\LabVIEW\Source Code\TestExec.llb\Full UI - Top-Level VI.vi	<TestStand Public>\UserInterfaces\Simple\LabVIEW\Source Code\TestExec.llb\Simple OI - Top-Level VI.vi
LabWindows/ CVI	<TestStand Public>\UserInterfaces\Full-Featured\CVI\Source Code\TestExec.prj	<TestStand Public>\UserInterfaces\Simple\CVI\Source Code\TestExec.prj
Microsoft Visual Basic .NET	<TestStand Public>\UserInterfaces\Full-Featured\VB.Net\Source Code\TestExec.VS2010.vbproj	<TestStand Public>\UserInterfaces\Simple\VB.Net\Source Code\TestExec.VS2010.vbproj

TestStand installs the source code files for the default user interfaces in the <TestStand>\UserInterfaces and <TestStand Public>\UserInterfaces directories. To modify the installed user interfaces or to create new user interfaces, modify the files in the <TestStand Public>\UserInterfaces directory. You can use the read-only source files for the default user interfaces in the <TestStand>\UserInterfaces directory as a reference. NI recommends that you track the changes you make to the user interface source code files so you can integrate the changes with any enhancements in future versions of the TestStand User Interfaces.

Related concepts:

- [TestStand Directory Structure](#)
- [Special Considerations for a LabVIEW Custom User Interface](#)
- [Required Visual Studio Components for Building Custom C# User Interfaces](#)
- [Caveats for Using the LabVIEW Example User Interfaces](#)

Required Visual Studio Components for Building Custom C# User Interfaces

To build custom C# user interfaces, you must install additional components with Visual Studio.

TestStand includes two directories containing example C# user interfaces:

- `<TestStand Public>\UserInterfaces\Full-Featured\CSharp\Source Code`
- `<TestStand Public>\UserInterfaces\Simple\CSharp\Source Code`

If you use the files in these directories as a starting point for creating custom user interfaces, you must install the Desktop development with C++ component for the version of Visual Studio you use. This component includes the MSVC Visual C++ build tools, which are required for building C# user interfaces in TestStand. You can install Desktop development with C++ during the initial Visual Studio installation or launch the Visual Studio installer to modify your existing installation.

Related concepts:

- [Example User Interfaces](#)
- [TestStand Directory Structure](#)

Caveats for Using the LabVIEW Example User Interfaces

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

- The `Load Top-Level VI.vi`, located in the `<TestStand Public>\UserInterfaces\Full-Featured\LabVIEW\Source Code` directory, launches the LabVIEW user interface in the context of a LabVIEW project.
- If you use the files located in the `<TestStand Public>\UserInterfaces\Full-Featured\LabVIEW\Source Code` or `<TestStand Public>\UserInterfaces\Simple\LabVIEW\Source Code` directories as a starting point for creating a user interface, NI recommends that you always run the user interface in the context of its LabVIEW project to avoid name

collisions. If you expect the user interface to run outside the context of the LabVIEW project, you must modify the names of the VIs in the user interface to ensure the names will not collide with the names of code module VIs or its subVIs.

- A LabVIEW project library provides a namespace for the LabVIEW simple and full-featured user interface examples. If you load any VI from the user interface by name, you must update the code that loads the VI to include the namespace.
- When you run the `TestExec.exe` build specification to generate an executable, LabVIEW prefixes VIs listed under the Dependencies node in the LabVIEW project with `Simple UI - or Full UI -` for the simple and full-featured user interfaces, respectively, to avoid name collisions. If you load any VI from the user interface by name, you must update the code that loads the VI to include VI prefixes. Complete one of the following options for all dynamic calls listed under the Dependencies node, including VIs that you call from top-level files but that you have not explicitly added to the My Computer target in the LabVIEW project, to resolve this issue:
 - Add the `Simple UI - or Full UI -` prefix to the path you pass to the Open VI Reference function when the caller VI runs in the LabVIEW RTE. You can use the Application:Kind property in LabVIEW to determine whether the VI is running in the development system or the LabVIEW RTE.
 - Add the VIs listed under the Dependencies node to the My Computer target to prevent the name change when you build the executable. LabVIEW prefixes only the files listed under the Dependencies node. NI recommends that you prefix these VIs manually to avoid potential name collisions that might prevent the user interface from running VIs that have conflicting names.



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

Related concepts:

- [Example User Interfaces](#)
- [TestStand Directory Structure](#)

Sequence File Translators

TestStand uses custom sequence file translators to load test description files saved in a custom format, such as text or XML. The translator reads the contents of the custom sequence file, translates the content to a TestStand sequence file, and opens the TestStand sequence file in the TestStand Sequence Editor or a TestStand User Interface. A custom sequence file translator can use predefined step types to simplify the mapping of common operations the custom file format defines to TestStand steps in sequence files.

Within the sequence editor or user interface, you can perform all typical operations TestStand sequence files support, such as executing and debugging sequences, diffing files, adding custom sequence files to workspaces, and deploying custom sequence files. However, you cannot automatically save changes you make to the sequence file in the sequence editor or user interface back to the custom sequence file format. You must make all changes to the custom sequence file directly.

You can create sequence file translators in various development environments, use versioning schemes with custom files, and deploy translators with TestStand. Refer to the `<TestStand Public>\Examples\Fundamentals\Sequence File Translators - Examples` directory for example custom sequence files and translators.



Note A sequence file translator can cause the TestStand UI in the sequence editor or in a user interface to hang if the translator launches a dialog box that is modal to the UI and if the translation occurs from within an execution. To prevent this behavior, avoid launching modal dialog boxes during translation, or ensure that you never translate a file from an execution. A dialog box is modal if you call the `Engine.NotifyStartOfModalDialogEx` method or if you use the `Engine.AppMainHwnd` property to specify the owner window for the dialog box. Translation occurs within an execution if you call the `Engine.GetSequenceFileEx` method in a sequence or if you execute a Sequence Call step that calls a custom sequence file.

Sequence File Translator Functions

A TestStand sequence file translator DLL must export and implement the following C callback functions. When a translator DLL does not export all the callback functions, TestStand does not load the DLL.

- CanTranslate
- GetDescription
- GetExtension
- GetFileFormatVersion
- GetFileVersion
- GetTranslatorCount
- IsCurrentFileVersion
- TranslateToSequenceFile

Error Handling

Each callback function contains three parameters for error handling—an error code, an error string, and the maximum length for the error string. When an error occurs within a callback function, set the error code to a non-zero value. When the value you assign is a TestStand error code, TestStand uses the standard error code description. The callback function can copy additional error details to the standard error message string. However, the callback must not exceed the specified number of bytes for the maximum length for the error message. TestStand uses the error message string the callback function specifies.



Note Avoid returning errors from C callback functions to TestStand, which might not properly handle the errors. Refer to the example translator projects in `<TestStand Public>\Examples\Fundamentals\Sequence File Translators - Examples` for examples of how to return errors from within callback functions.

Related concepts:

- [Creating a Sequence File Translator DLL](#)
- [TestStand Directory Structure](#)
- [Sequence File Translators](#)

- [Deploying TestStand Systems](#)

Creating a Sequence File Translator DLL

TestStand can load custom sequence files if an existing translator can read and convert the file into a TestStand SequenceFile object. Translators are Microsoft Windows DLLs you can create that export required C callback functions TestStand uses to translate files.

You can create custom sequence file translators in any development environment that can create a Windows DLL with the required C callback functions. NI recommends using the translator examples written in LabVIEW, LabWindows/CVI, and Microsoft Visual C++ as a guide. Each example in the `<TestStand Public>\Examples\Fundamentals\Sequence File Translators - Examples` directory includes a template project, which contains source code with empty callback functions you must export from the translator DLL. You must add the necessary code to the required callbacks to ensure that the translator properly integrates with TestStand.

When an application loads the TestStand Engine, TestStand loads the DLLs that export the required callback functions from the `<TestStand>\Components\Translators` directory and the `<TestStand Public>\Components\Translators` directory. When you run a TestStand sequence file translator example, located in the `<TestStand Public>\Components\Translators` directory, the example copies the example translator DLL to the `<TestStand Public>\Examples\Fundamentals\Sequence File Translators - Examples` directory, and you must restart the application before you can use the translator.

To create new translator DLLs, add the project for the translator in the `<TestStand Public>\Components\Translators` directory and ensure that the project saves the DLL to the `<TestStand Public>\Components\Translators` directory. Storing new and customized files in the `<TestStand Public>` directory ensures that newer installations of the same version of TestStand do not overwrite the customizations and ensures that uninstalling TestStand does not remove the files you customize.

A translator DLL can contain one or more translators. When TestStand loads a

translator DLL, TestStand uses the callback functions of the DLL to obtain information about the translators the DLL contains. TestStand calls the CanTranslate callback function to determine whether the DLL contains a translator that recognizes a file. The callback returns the index of the translator that recognizes the file after examining the extension of the file and the contents of the file, typically the file header. Most of the callback functions the translator DLL implements contain an **index** parameter, which references a specific translator in the DLL that must operate on a file.

Related concepts:

- [Example Sequence File Translators](#)
- [TestStand Directory Structure](#)

Example Sequence File Translators

The LabVIEW, LabWindows/CVI, and Microsoft Visual C++ example projects demonstrate how to build translator DLLs and provide guidance for developing translators. The examples illustrate two simple translators for each development environment that use the TestStand API to convert sample test descriptions in XML and ASCII text formats into TestStand sequence files. The example translators for each file format produce the same TestStand sequence file.

The sample test descriptions specify steps that perform a calculation, display the result of the calculation in a graph, compare the result with an expected value, and display a message that indicates whether the test passed or failed. The translation from the example format into a sequence file involves adding steps and local variables to a sequence in a new sequence file object and configuring the steps to perform the required operations. The translators also use a custom step type TestStand loads from a type palette file that you must place in the <TestStand Public>\Components\TypePalettes directory.

Complete the following steps to use an example.

1. Open the `TextTranslator` or `XMLTranslator` directory for one of the examples in the <TestStand Public>\Examples\Fundamentals\Sequence File Translators - Examples directory.
2. Copy the type `NI_ExampleTranslatorTypes.ini` file from the

<TestStand Public>\Examples\Fundamentals\Sequence File Translators - Examples directory to the <TestStand Public>\Components\TypePalettes directory.

3. Open and review the project in the development environment for the example.
4. When you make any changes to the project, rebuild the project to update the translator DLL. Copy the translator DLL into the <TestStand Public>\Components\Translators directory.
5. Launch the TestStand Sequence Editor or a TestStand User Interface to load the translator DLLs.
6. Select **File»Open File** and select `SampleTestFile.xml`, `SampleTestFile.lvtf` for the text version of the file for LabVIEW, `SampleTestFile.cvitf` for the text version of the file for LabWindows/CVI, or `SampleTestFile.vctf` for the text version of the file for Microsoft Visual C++.
7. Review the translated sequence file.
8. Launch an execution using the `MainSequence` in the sequence file.

Related concepts:

- [Sequence File Translators](#)
- [TestStand Directory Structure](#)

Versioning Translators and Custom Sequence Files

When you edit a custom sequence file format, you can increment the following information:

- **Version number of the file format**—Identifies the structure and syntax of the file.
- **File version number**—Identifies the revision of the contents of the file.

When the contents of a custom sequence file includes a file format version number, a translator can read files with the current file format and files with an earlier file format, and the translator can identify newer file formats it does not support. When a translator callback accesses the contents of the file, the translator ensures that it can support the file format version. For example, the `CanTranslate` callback uses the version number to determine whether the translator can load the file. In addition, TestStand displays the return value from the `GetFileFormatVersion` callback in reports the Workspace Documentation tool creates.

When the contents of a custom sequence file includes a file version number or revision, implement the translator to assign the version to the `PropertyObjectFile.Version` property in the `TranslateToSequenceFile` callback and return the version in the `GetFileVersion` callback to ensure that the Sequence File Properties dialog box displays the file version number and the Sequence File Documentation and Workspace Documentation tools display the file version number in reports you create.

When the file formats between version numbers differ significantly, consider creating two translators in a single DLL or a separate translator in two DLLs to simplify the code necessary to translate each file format. When files contain header fields that identify the file format and the `CanTranslate` callback uses these fields, ensure that using two translators does not affect the performance of opening files in TestStand.

Deploying Translators and Custom Sequence Files

When you place a custom sequence file translator DLL and its support files in the `<TestStand Public>\Components\Translators` directory on the computer you use to build the deployment, the TestStand Deployment Utility automatically includes these files in the deployment when you enable the Deploy Files in TestStand Public Directories option on the System Source tab of the deployment utility. Alternatively, you can add the translator files to the workspace and set the target destination directory for the files to the `<TestStand Public>\Components\Translators` directory.

When the deployment utility analyzes a TestStand sequence file, the utility completes the following steps to ensure that TestStand can locate the code module on the computer where you deploy the files.

1. Locates the code modules the steps in the sequence file call.
2. Adds the code module files to the deployment.
3. Changes absolute path references in sequence files to relative path references.

You can add custom sequence files to the workspace the deployment utility uses to build a deployment. The deployment utility must load and translate custom sequence files to locate the code modules the steps in the sequence file call. However, the deployment utility does not modify the paths in the custom sequence file and returns

a warning when the deployment utility cannot ensure that TestStand can locate the code module on the computer where you deploy the files. You must fix the paths on the computer you use to build the deployment, or you must fix the paths on the target computer after deployment.

Related concepts:

- [Creating a Sequence File Translator DLL](#)
- [TestStand Directory Structure](#)
- [Deploying TestStand Systems](#)

Creating String Resource Files

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

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

Select **Configure»Station Options** to launch the Station Options dialog box and click the Localization tab to change the current language setting.

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



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

Related concepts:

- [TestStand Directory Structure](#)

String Resource File Format

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

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

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

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

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

```
[category1]
tag1 Line0001 = "This is the first sentence of a long "
tag1 Line0002 = "paragraph. This is the second sentence."
```

You can use the escape codes in the following table to insert unprintable characters.

Table 42. Escape Code Descriptions

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

Using Enumerations in TestStand

The TestStand enumeration data type provides several advantages over the traditional alternative of using a number to represent an enumerated data type, including range checking, type safety, and the ability to extract the enumerator name as a string. To provide complete backward compatibility, all TestStand adapters provide full support for passing numbers for enumeration parameters.

TestStand enumerations attempt to provide an experience that as closely as possible mimics the behavior of enumerated data types in common modern programming languages. You typically interact with enumerated data types using their enumerators at edit time, while operations are generally based on the underlying number at run time. There are subtleties that arise from the fact that the distinction between edit time and run time in TestStand is not clear-cut. Refer to the TestStand Enumerations Reference Guide for a detailed explanation of the behavior of TestStand enumerations.

Related concepts:

- [TestStand Enumerations Reference Guide](#)
- [Getting Started Creating a TestStand Enumeration](#)

Getting Started Creating a TestStand Enumeration

Creating an Enumeration Data Type and Defining the Enumerators

To create a TestStand enumeration, go to the Types View in the TestStand Sequence Editor. In a sequence file of your choosing, go to **Custom Data Types** and right click to insert a type. From the context menu, select **Insert Custom Data Type** and choose **Enumeration**. Name the newly created data type `Color`, for example. Note that the current state of the type `Color` is `{Invalid} [] (0)`. The `Color` type currently has no enumerators, so its default value is invalid. This is one example of how a TestStand enumeration can end up in an invalid state.

Click the **Edit Enumerators** button next to `Color` to launch the Edit Enumerators dialog box. In the table control, define the following three enumerators:

Red	255
Blue	65280
Green	16711680



Note For convenience, you can enter the above numbers using their hexadecimal values: `0xff`, `0xff00`, and `0xff0000`.

Click **OK** to commit these changes and close the dialog box. Notice that `Color` is no longer invalid and has a default value of `Red`. The underlying numeric value of `255` is displayed in parentheses for convenience.

Since hexadecimal is more convenient for this example, change the numeric format. Right click on the type definition. In the context menu, choose **Numeric Format** to launch the **Numeric Format** dialog box. Change the type to `Hexadecimal` and click **OK**. Note that the number in parentheses is now displayed in hexadecimal format.

Range Checking: Strict and Loose Enumerations

TestStand enumerations can be either strict or loose. A loose enumeration can take on any numeric value. Enumerations in C, C++, and C# can be described as loose in this sense. A strict enumeration can take on only those values specified by its enumerators. Attempting to set a strict TestStand enumeration to a value that does not correspond

to its enumerators results in an error. Use the Edit Enumerators dialog box to specify that an enumeration data type is strict or loose.

Flags Enumerations and Bitwise OR Semantics

TestStand enumerations may be designated as "flags." This is conceptually similar to adding the `[Flags]` attribute to an enumeration in .NET, although the functionality is different. In TestStand, flags enumerations support bitwise semantics to specify combinations of enumerators. Non-flags enumerations do not support these semantics. Use the Edit Enumerators dialog box to specify that an enumeration data type is a flags enumeration.

Creating an Enumeration Instance

In the Variables View for a sequence, under Locals, right click to launch the context menu. Choose **Insert Local»Type»Color** to create an instance of the Color enumeration. The instance has the same default value and numeric format as the type definition. You can edit the value of this instance just like the type definition by selecting one of the enumerators from the value drop-down or by launching the Edit Enumeration Value dialog box.



Note There is no option to **Insert Local»Enumeration** in the Variables View because all enumerations must be instances of a custom enumeration type. Without a type definition, an enumeration would have no enumerators, making it no different than a number.

TestStand Enumerations Reference Guide

Enumerations as Numbers

Enumeration data types have numeric format and representation just like TestStand numbers. Both can be set on the type definition in exactly the same way they are set for numbers. The numeric format may be different for type instances than it is for the definition. The numeric representation is a property of the type and cannot be changed on individual instances.

Flags enumerations with the default 64-bit float representation must be integers in the range $[0, 2^{32} - 1]$ inclusive. You can set non-flags enumerations with the default representation to any value, although you should use non-integer values with caution since most code modules treat enumerations as integers.

You can convert a TestStand enumeration to a number using `GetValNumber()`, `GetValInteger64()`, or `GetValUnsignedInteger64()` as appropriate with the `PropOption_CoerceFromEnum` option. Alternatively, you can use the `Float64()`, `Int64()`, or `UInt64()` expression functions. You can assign a number to an enumeration using `SetValNumber()`, `SetValInteger64()`, or `SetValUnsignedInteger64()` as appropriate with the `PropOption_CoerceToEnum` option. For further details, see the [Enumerations as PropertyObjects](#) and [Enumerations in TestStand Expressions](#) sections.

Related concepts:

- [Enumerations as PropertyObjects](#)
- [Enumerations in TestStand Expressions](#)

Enumerations as PropertyObjects

TestStand enumerations are `PropertyObjects` just like other TestStand data types. As such, you can invoke `PropertyObject` methods and properties on enumerations. In most cases, the extension of existing methods to TestStand enumerations is obvious, but cases specifically related to type or value require further explanation.

<Enumeration.Type.ValueType> — Returns `PropValType_Enum`.

<Enumeration>.SetValString() — `SetValString()` fails with a type mismatch error unless you pass `PropOption_CoerceToEnum`. With the `PropOption_CoerceToEnum` option, `SetValString()` attempts to set the enumeration instance to the enumerator matching the string value passed. If the string value matches one of the enumerators, the call succeeds and both the enumerator and numeric values of the enumeration instance are updated. For flags enumerations, the string can be any combination of enumerators combined with bitwise OR, including the empty string. Passing a string that does not correspond to the enumerators results in a runtime error.

<Enumeration>.GetValString() — `GetValString()` fails with a type mismatch error unless you pass `PropOption_CoerceFromEnum`. With the

`PropOption_CoerceFromEnum` option, `GetValString()` returns a string representation of the enumeration. If the enumeration instance has enumerator information, the enumerator name is returned. For flags, this may be multiple enumerators combined with bitwise OR.

If the enumeration instance has no enumerator information (i.e. because it is loose and set to a value that does not correspond to its enumerators), the behavior depends on the options you pass. If you pass `PropOption_CoerceFromNumber`, the numeric value is converted to a string and returned. For flags enumerations, this may be a combination of enumerators combined with bitwise OR along with whatever number is left over. If you do not pass `PropOption_CoerceFromNumber`, `GetValString()` returns an empty string. Generally, `PropOption_CoerceFromNumber` is appropriate when the purpose of the string is for display. `PropOption_CoerceFromNumber` is not usually appropriate for data persistence applications.

Calling `GetValString()` on an invalid enumeration results in a runtime error.



Note `PropOption_Coerce` includes both `PropOption_CoerceFromEnum` and `PropOption_CoerceFromNumber`.

Property Options(s)	Enum State	Result
None	Any	Runtime error
<code>CoerceFromEnum</code>	Has enumerator info	Enumerator(s) represented as string.
<code>CoerceFromEnum</code>	No enumerator info	Empty string.
<code>CoerceFromEnum CoerceFromNumber</code>	Has enumerator info	Enumerator(s) represented as string.
<code>CoerceFromEnum CoerceFromNumber</code>	No enumerator info	Numeric value converted to a string. For flags enumerations, could be some combination of enumerators combined with the OR operator with left-over number.
Any	Invalid	Runtime error

<Enumeration>.SetValNumber() — SetValNumber() fails with a type mismatch error unless you pass PropOption_CoerceToEnum. With the PropOption_CoerceToEnum option, SetValNumber() attempts to set the enumeration instance to the specified value. If the number matches one of the enumerators, the call succeeds. For flags enumerations, the number can correspond to any combination of enumerators combined with a bitwise OR, including 0, which indicates no enumerators. The enumerator info is cleared unless the CoerceToString option is passed. If CoerceToString is passed, the enumerator information will be updated to match the numeric value. If there are multiple enumerators or combinations of flags that can represent the specified numeric value, a valid combination is chosen arbitrarily.

If the number does not correspond to the enumerators, the behavior depends on whether the enumeration is strict or loose. If the enumeration is loose, the number is updated to the specified value and the enumeration information is cleared. If the enumeration is strict, SetValNumber() fails with a runtime error.

Property Options(s)	Strict/ Loose	Number In Range	Behavior
None	Any	Any	Runtime error
CoerceToEnum	Any	In Range	Numeric value set. Enumerator information cleared.
CoerceToEnum CoerceToString	Any	In Range	Numeric value set. TestStand engine sets enumerator(s) matching specified value. If ambiguous, choice is arbitrary.
CoerceToEnum	Strict	Out of Range	Runtime error. (CoerceToString has no effect.)
CoerceToEnum	Loose	Out of Range	Numeric value set. Enumerator information cleared. (CoerceToString has no effect.)

<Enumeration>.GetValNumber() — GetValNumber() fails with a type mismatch error unless you pass PropOption_CoerceFromEnum. With the PropOption_CoerceFromEnum option, GetValNumber() returns the numeric value of the enumeration. Calling GetValNumber() on an invalid enumeration results in a runtime error.

GetValInteger64() and GetValUnsignedInteger64() work similarly for enumerations with the associated representations. As with TestStand numbers, passing PropOption_CoerceFromNumber coerces the numeric representation if

necessary.

Enumerations in TestStand Expressions

Casting Functions

`Float64()` — Equivalent to calling `GetValNumber()` with `PropOption_Coerce.Int64()` — Equivalent to calling `GetValInteger64()` with `PropOption_Coerce.UInt64()` — Equivalent to calling `GetValUnsignedInteger64()` with `PropOption_Coerce.Str()` — Equivalent to calling `GetValString()` with `PropOption_Coerce`.

Operators

Equality (==) and Inequality (!=) — To enforce type safety, enumerations can be compared only to other enumerations of the same named type. The equality and inequality operations compare the numeric values of the enumeration instances, ignoring any enumerator information. For example, if both `Scarlet` and `Red` have the numeric value of 2, then an enumeration set to `Scarlet` will equal an enumeration set to `Red`. (This behavior corresponds to the behavior of enums in C#, C, and C++.)

Assignment (=) — To enforce type safety, enumerations can be assigned only to other enumerations of the same named type.

Logical NOT (!) — Supported for flags enumerations only. Returns true if the numeric value is zero, false otherwise. Use logical NOT to check if any flags are set.

Bitwise AND (&), OR (|) and XOR (^) — Supported for flags enumerations only. To enforce type safety, enumerations can be combined only with other enumerations of the same named type. The TestStand engine performs the specified operation on the numeric values of the two operands, then constructs a new instance of the enumeration type and sets the value.



Note This behavior can result in a runtime error for strict enumerations if the resulting value is out of range.

Bitwise NOT(~) — Supported for flags enumerations only. The TestStand engine

performs the bitwise NOT operation on the numeric value of the operand, then constructs a new instance of the enumeration type and sets the value.



Note This behavior can result in a runtime error for strict enumerations if the resulting value is out of range.

Unsupported Operations (<, >, +, -, *, /, %, +=, -=, *=, /=, %=, &&, ||, <<, >>) — These operations are not supported for enumerations. To perform these operations, you must first convert the enumeration to a number.

Enumeration Constants

Use the `Enum()` expression function or the `Enums` keyword to specify an enumeration constant in an expression.

Enums keyword — Use the syntax `Enums.<TYPE>.<ENUMERATOR>` to specify an enumeration constant in a TestStand expression. For example, `Enums.Color.Red` specifies the `Red` value from the `Color` type in the Getting Started Creating a TestStand Enumeration section.

Enum(String type, [String|Number] value) — Returns an instance of an enumeration with the specified type and value. If value is a string, the enumerator and number are set as if by calling `SetValString()`. Passing an out of range value results in an error. If value is a number, the value is set as if by calling `SetValNumber()`, `SetValInteger64()`, `SetValUnsignedInteger64()` as appropriate for the representation of the enumeration type. Passing an out of range value to a strict enumeration results in a runtime error.

Related concepts:

- [Getting Started Creating a TestStand Enumeration](#)

Using Enumerations with the TestStand Adapters

Sequence Call Adapter

The type of variable passed for an enumeration parameter must exactly match the parameter type in the sequence being invoked.

C/C++ DLL and LabWindows/CVI Adapters

There are two ways to pass enumeration parameters to the C/C++ DLL and LabWindows/CVI adapters:

- Pass a number specifying the numeric value of the parameter.
- Pass a TestStand enumeration instance. (Requires TestStand 2016 or later.)

To pass a TestStand enumeration, choose **Enumeration** for the category of the parameter in the module panel. Then select a data type from the drop-down menu for the Enum Type control.

When specifying the expression for the parameter value, you can select an enumerator in the drop-down menu, manually enter an enumerator constant, or specify an expression that evaluates to the required enumeration data type.

LabVIEW Adapter

You can create an Enum data type that maps to the LabVIEW enumerated parameter from an enumeration, ring control, or indicator by clicking on the **Create/Update Custom Data Type** button located in the Type column of the VI Parameter table next to any LabVIEW enumerated parameter or an enumerated array parameter.

This launches the Create/Update Custom Data Type from Enum Dialog Box, where you can create a custom Enum data type or update an existing Enum data type to match the LabVIEW parameter.

You can create an instance of the Enum data type and specify that variable in the Value column of the VI Parameter table.

When specifying string or Enum variables as the value of an enumerated parameter, the following table describes the behavior:

TestStand Type	Variable Passed to a LabVIEW Input Parameter	Variable Receiving Data from a LabVIEW Output Parameter
String	If the variable's value exactly matches any of the LabVIEW enumerated parameter's elements, the numeric	If the numeric value received from LabVIEW matches any of the enumerated parameter's elements, then the matching element's

TestStand Type	Variable Passed to a LabVIEW Input Parameter	Variable Receiving Data from a LabVIEW Output Parameter
	value of the element is passed to LabVIEW. If there is no match, you will get a run-time error. Assigning a duplicate (more than one element with the same name) enumerator name will result in an error.	name is stored in the string. If the numeric value does not match any element, its string representation surrounded by angular brackets is stored. Assigning a duplicate enumerator value to the string variable will result in an error.
CustomEnum data type	The elements of a TestStand Enum data type should match the LabVIEW parameter's enumerated elements exactly (both name and value). Any mismatch will result in an error. If the type matches, then the numeric value of the selected enum element is passed to LabVIEW.	The elements of a TestStand Enum data type should match the LabVIEW parameter's enumerated elements exactly (both name and value). Any mismatch will result in an error. If the TestStand Enum data type is Strict, then the numeric value received from LabVIEW must match one of the enumerator elements. If there is no match, you will get a run-time error.

.NET Adapter

There are three ways to pass enumeration parameters to the .NET adapter:

- Pass a TestStand string specifying the enumerator.
- Pass a TestStand number specifying the numeric value of the parameter.
- Pass a TestStand enumeration instance. (Requires TestStand 2016 or later.)

If passing a TestStand enumeration for a .NET enumeration parameter, the definition of the TestStand enumeration type should match the definition in the .NET code module. Specifically, the TestStand enumeration should define the same set of enumerators with the same numeric values as the code module. In addition, the numeric representation must match with the same requirements that apply to TestStand numbers. The name of the type is not important.

At edit time, the module panel will indicate if the definition of the TestStand enumeration specified does not match the definition in the code module. At run time, limited type checking is performed:

- If enumerator information is present in an enumeration instance used as an input parameter, the enumerators used by the instance must exist in the code module.

Furthermore, the numeric value must match the value from the code module. (For flags enumerations, only the combined value is checked.)

- For both input and output parameters, the numeric representation of the TestStand enumeration must be compatible with the definition in the code module.



Note Use the TestStand Sequence Analyzer to find enumeration type mismatch errors in .NET adapter steps.

ActiveX/COM Adapter

There are two ways to pass enumeration parameters to the ActiveX/COM adapter:

- Pass a number specifying the numeric value of the parameter.
- Pass a TestStand enumeration instance. (Requires TestStand 2016 or later.)

If passing an enumeration for an ActiveX/COM enumeration parameter, the definitions of the TestStand enumeration type should match the definition in the type library for the code module. Specifically, the TestStand enumeration should define the same set of enumerators with the same numeric values as the code module. In addition, the numeric representation must be the default. The name of the type is not important.

At edit time, the module panel will indicate if the definition of the TestStand enumeration specified does not match the definition in the code module. At run time, limited type checking is performed:

- If enumerator information is present in an enumeration instance used as an input parameter, the enumerators used by the instance must exist in the code module. Furthermore, the numeric value must match the value from the code module. (For flags enumerations, only the combined value is checked.)
- For both input and output parameters, the numeric representation of the TestStand enumeration must be compatible with the definition in the code module.



Note

- Use the TestStand Sequence Analyzer to find enumeration type mismatch errors in COM/ActiveX adapter steps.
- Many ActiveX/COM APIs expose enumerations in their type libraries, but still specify the long data type in method prototypes that expect these enumeration parameters. To pass a TestStand enumeration to such a method, convert it to a number using the appropriate expression function: `Float64()`, `Int64()`, or `UInt64()`.

Enumeration Options

Strict vs. Loose Enumerations

A TestStand enumeration can be either strict or loose. A loose enumeration can take on any numeric value. Enums in C, C++, and C# can be described loose in this sense. A strict enumeration can take on only those values defined by its enumerators. Attempting to set a strict TestStand enumeration to a value that does not correspond to its enumerators results in an error. By default, TestStand enumerations are strict.

Flag Enumerations

TestStand enumerations may be designated as “flags.” This is conceptually similar to adding the `[Flags]` attribute to an enum in .NET, although the functionality is different. In TestStand, flags enumerations support bitwise semantics to specify combinations of enumerators. Non-flags enumerations do not support these semantics.

Enumerations and Type Merging

When updating an enumeration instance, the enumerator information is never changed. If the enumerator exists in the updated type definition, the numeric value of the instance is updated to match the new type definition. If the enumerator does not exist, the instance being updated is left unchanged, but is now invalid.

This behavior is analogous to behavior in C, C++, and C#. In these languages, changing the value of an enumerator in the enumeration definition implicitly changes the

numeric value of the enumerator everywhere it is used in source files. Deleting or renaming an enumerator in the enumeration definition causes dependent code not to compile until it has been fixed.



Note When you rename an enumerator using the Edit Enumerators dialog box in the types view, loaded instances of the type will be renamed to match. However, renaming operations cannot propagate later via type merging. Further propagation will behave as if the old enumerator had been deleted and a new one added.

Invalid TestStand Enumerations

Most TestStand API prevents enumeration instances from becoming invalid. Nevertheless, changing the type definition can make existing instances invalid. For example, if an enumerator is deleted from the type definition, any existing instances set to that enumerator become invalid. Changing the flags and strict/loose setting can similarly render existing instances invalid.

Assigning new values to invalid enumerations is possible. Conceptually, this behavior is similar to initializing an uninitialized variable in a language like C++. It does not matter what the value used to be after you set the new one.

In most cases, attempting to obtain or use the value of an invalid enumeration results in a runtime error. Methods such as `GetValString()` and `GetValNumber()` fail when invoked on invalid enumerations. TestStand expression operations, such as the equality operator (`==`), result in runtime errors for invalid enumerations as well. Similarly, passing an invalid enumeration to a code module via the adapters results in a runtime error.

One exception to this rule is `PropertyObject.GetDisplayNames`. When you call `PropertyObject.GetDisplayNames` on an invalid enumeration, the output value `DisplayNameString` parameter indicates the internal state of the enumeration and that it is invalid.

Another exception is serialization methods. Property object files and sequence files can be saved or read even if they contain invalid enumerations.

`PropertyObject.GetXML` or `PropertyObject.SetXML` work with invalid enumerations if

you pass the `XMLOption_AllowInvalidObjects` option.

Developing TestStand Systems

Refer to the NI TestStand Advanced Architecture Series for more advanced concept and architecture information. This series is a suite of documents TestStand architects and developers created to provide detailed information for experienced TestStand users with complex projects.

Visit ni.com/info and enter the Info Code `BldTestSys` to access the NI Fundamentals of Building a Test System guides for more information about the recommended process for building test systems.

Visit ni.com/info and enter the Info Code `TestMgtSWDevGd` to access the NI Test Management Software Developers Guide for more information about developing test systems.

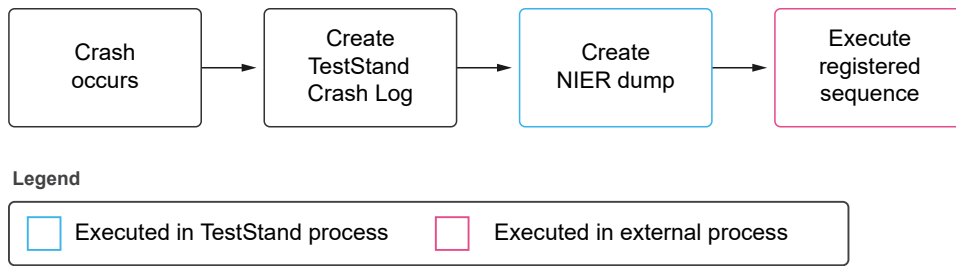
Crash Recovery for TestStand Applications

TestStand has the option to generate a log containing execution information and use the NI Error Reporting Service, NIER, to automatically generate a crash dump file. You can use a crash dump file to diagnose the root cause of a crash. In cases where the crash is in an NI product, you can send the crash dump file to NI support engineers for diagnosis. TestStand also offers a way to register a custom sequence to execute in an external process in response to a crash.

TestStand Crash Log—When a crash occurs TestStand generates a log file in the directory `<TestStand_LocalAppData>\CrashLogs`. Only the last 100 crash logs are saved, after which the earliest log file will be deleted and a new log file is created.

NIER Crash Dump—Using the same error reporting service as LabVIEW, TestStand generates a NIER crash dump in `<TestStand_LocalAppData>\NIERdump`.

The following figure shows the sequence that occurs when a crash occurs in a TestStand application.



TestStand Crash Log and Dump File

TestStand creates the following files after a TestStand application crash:

- Dump file in <TestStand_LocalAppData>\NIERDump directory
- TestStand specific log file in <TestStand_LocalAppData>\CrashLogs directory

You can use the configuration file to specify settings for the dump and TestStand specific log file. The configuration file is an .ini file in the same directory as the TestStand application. The name of the configuration file must be same as the executable name of the TestStand application.

In the configuration file, create a section with the same name as the name of the executable. Use the following tokens and values to specify the options:

Token Name	Value	Purpose
TSLogType	None	Do not create TestStand crash log. By default, TestStand crash log is created.
NIER	False	Do not create NIER crash dump. By default, NIER crash dump is created.
NIERDumpType	Full	Create full (instead of mini) NIER crash dump. By default, mini dump file is created.
LogVariables	True	Enable logging of Locals, Parameters, and FileGlobals in the TestStand specific log file. By default, Locals, Parameters, and FileGlobals are not logged in the TestStand log file.



Note Crash log and dump file are not sent to any server.



Note Crash log and dump file are not generated if the application crashes

before creating an instance of the TestStand Engine.

The different configuration options combined with the number of different environments supported by TestStand affects how the crash recovery feature operates. Below is a table of different behaviors produced by the crash recovery feature in different scenarios:

Behavior	Using TestStand 2017 in LabVIEW ADE	Using TestStand 2017 in all other scenarios	Using older version of TestStand in LabVIEW ADE	Using older version of TestStand in LabVIEW-built executable	Using older version of TestStand in all other scenarios
TestStand crash log	Created	Created	Not created	Not created	Not created
NIER dump	Created by TestStand	Created by TestStand	Created by LabVIEW	Created by LabVIEW	Not created
Send data to NI server	NIER dump is sent to NI server on user consent.	No data is sent to NI server.	NIER dump is sent to NI server on user consent.	No data is sent to NI server.	N/A
TSLogType=none in configuration file	TestStand log is not created. NIER dump is created.	TestStand log is not created. NIER dump is created.	N/A	N/A	N/A
NIER=false in configuration file	TestStand log is created. NIER dump is not created.	TestStand log is created. NIER dump is not created.	NIER dump is not created.	NIER dump is not created.	N/A

Restrictions of using the feature

- Code modules should not register to functions like SetUnhandledExceptionFilter, set_invalid_parameter_handler, and set_purecall_handler to handle application crash.
- Crash due to out of memory is not supported.

Registering a Crash Recovery Sequence

You can register sequences to respond to a crash in your TestStand application to return any hardware or other system assets into a good state. The sequences are executed by an external process and run without a process model or a logged-in user. Since multiple crash recovery sequences can be registered, the order in which the sequences will be executed is the order in which they are registered.

Any errors encountered in executing crash recovery sequences are recorded in a log file, and a dialog displays the location of the log file. Similar to the main application TestStand crash log, this log file stores the errors from a maximum of 100 crash callbacks.

To register a sequence for execution in the case of a crash you must call `Engine.RegisterSequenceToExecuteOnCrash` with the sequence name and file path to be executed. This function returns a unique ID that can be used to unregister the sequence later with a call to `Engine.UnregisterSeuquenceToExecuteOnCrash`.

TestStand Tools and Utilities

Refer to the topics in this section for information on TestStand tools and utilities provided by NI, and their usage in TestStand projects.

TestStand Sequence Editor

The TestStand Sequence Editor is a development environment in which you create, edit, execute, and debug sequences and the tests sequences call. Use the sequence editor to access all TestStand features, such as step types and process models. The sequence editor also includes the following debugging tools:

- Setting breakpoints
- Stepping into, out of, or over steps
- Tracing through program executions
- Displaying variables
- Monitoring variables, expressions, and output messages during executions
- Performing static analysis of sequence files to locate errors and enforce coding guidelines

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

The sequence editor contains the following advanced editing features:

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

- Find and replace
- Integrated sequence file differ
- User management

You can customize the layout of panes and tabs to optimize development and debugging tasks. You can also interactively customize the menus, toolbars, and keyboard shortcuts.

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

Related concepts:

- [Executing a Sequence Directly](#)
- [Debugging Executions](#)
- [Step Types](#)
- [Process Models](#)
- [Expressions](#)
- [Adding Users and Setting Privileges](#)

Customizing TestStand Sequence Editor Toolbars and Menus

The default toolbars and menus in the TestStand Sequence Editor are fully customizable.

You can rearrange top-level menus and toolbar items within a toolbar by holding down the <Alt> key and dragging the menu or toolbar command to a new location.



Note Select View»Reset UI Configuration at any time to reset the user interface to the default configuration, which reverts all modifications made to docking panes, menus, and toolbars.

The Customize dialog box contains the following tabs:

- **Toolbars**—Controls the visibility of toolbars, creates new toolbars, and resets the

commands available in toolbars.

- **Commands**—The available commands for menus and toolbars. You can drag and drop commands from the Commands tab to menus or toolbars to the menus and toolbars in the main application. You can also remove a command from a menu or toolbar by dragging the command from the main application and dropping it in the Commands control.

You can also drag and drop commands between toolbars and menus to rearrange the order of toolbars and menus. Select **Rearrange** to launch the Rearrange Commands dialog box, in which you can add, delete, and rearrange the commands within menus and toolbars.

- **Options**—Specifies whether the sequence editor personalizes menus and toolbars, shows icons for menus and toolbars, shows toolbar tips, and if toolbar animation is enabled.

You can also click **Keyboard** in the Customize dialog box to launch the Customize Keyboard dialog box, in which you can customize the shortcut keys for all commands.



Note The Customize Keyboard dialog box does not allow you to assign a single key to multiple commands. However the default configurations assign the <F5> and <Ctrl-F5> keys to multiple commands, such as Run Test UUTs, Restart, and Continue. If you attempt to reassign these keys to another command, the Customize Keyboard dialog box unassigns only one of the default assignments.

When the Customize dialog box is visible, you can also right-click a menu, a menu command, or a toolbar command to display a context menu for customizing the appearance of a command.

Configuring Keyboard Shortcuts in the TestStand Sequence Editor

Complete the following steps to configure keyboard shortcuts or keyboard mappings in the TestStand sequence editor:

1. **View»Customize Toolbars/Menus** to launch the Customize dialog box.
2. Click **Keyboard** to launch the Customize Keyboard dialog box, in which you can customize the shortcut keys for all commands.



Note The Customize Keyboard dialog box does not allow you to assign a single key to multiple commands. However the default configurations assign the <F5> and <Ctrl-F5> keys to multiple commands, such as Run Test UUTs, Restart, and Continue. If you attempt to reassign these keys to another command, the Customize Keyboard dialog box unassigns only one of the default assignments.

Locking Toolbars, Menus, and Panes

You can configure the TestStand Sequence Editor to prevent a user from customizing the user interface. For example, you might want to prevent a user from rearranging the toolbars or panes within the application, or you might want to hide specific panes, buttons, and menus so the user cannot access specific features.

1. Click **Configure»Sequence Editor Options** to open the Sequence Editor Options dialog box.
2. Select the UI Configuration tab.
3. Select **Lock UI Configuration by Disabling the Following**, then select the check box for each element or operation you want to lock in the user interface.

TestStand Sequence Editor Keyboard Navigation

The TestStand Sequence Editor supports keyboard navigation between panes and windows, between panes in windows, between tabs in panes, and within tree and table controls.

Navigating Between Panes and Windows

Press <Ctrl-Tab> to display the Pane Navigation menu and hold the <Ctrl> key to prevent TestStand from closing the menu. The Pane Navigation menu contains a Panes list and a Windows list which represent all the panes and windows currently visible in the sequence editor.

While you hold the <Ctrl> key you can use the arrow keys or <Tab> and <Shift-Tab> to select a new pane or window to activate when you release the <Ctrl> key.

You can also use the keyboard to activate any pane through the View menu and any window through the Window menu.

Navigating Between Panes within a Window

Press the <Tab> key to navigate between panes in a window, such as the Sequences pane, Steps pane, and Variables pane.

Navigating within Panes

Press the <Tab> key to navigate between controls within a pane. If a pane contains tabs, when you navigate to the tab, use the left and right arrow keys to change the visible tab.

Navigating within a List

For list controls that contain multiple columns, such as the following controls, use the arrow keys to navigate between rows and columns:

- Variables list on the Variables Pane
- Parameters Table on a module tab
- TestStand Sequence Analyzer rules, files, and analysis modules
- TestStand File Diff and Merge utility

When you navigate to the plus or minus icon for an expandable row in the list, press <Space> or <Enter> to expand or collapse the row. You can also press the <+> or <-> keys on the keypad to expand or collapse a selected row, or press the <*> key on the keypad to expand the selected row and all items within the row. Press <Ctrl+> on the keypad to standardize the width of each column to the width of the widest column.

When you press <Enter> in a field of a Value column, the focus moves to the value field for the next row.

Press <Space> or <Ctrl-F4> to display the drop-down list for a combo box control, such as a Boolean value.



Note You cannot use the keyboard to access the buttons that appear within

columns, such as **Resize Array** and **Change Type**. When you edit expressions, use the context menu of the control to launch the Expression Browser dialog box and to check for errors.

Navigating with the Insertion Palette

Press the <Tab> key to navigate between the expand/collapse buttons, the Adapter list, the Step Types list, and the Templates lists. Use the up and down arrows to move within the lists. Use the left and right arrow keys to expand and collapse folders within the lists.

In the Templates list, you can also press the <+> or <-> keys on the keypad to expand or collapse a selected row, or press the <*> key on the keypad to expand the selected row and all items within the row.

Related concepts:

- [Expressions](#)

Drag and Drop Step Creation in the TestStand Sequence Editor

Dragging and Dropping Sequences to Create Sequence Call Steps

The TestStand Sequence Editor supports basic drag and drop operations like dragging and dropping of sequences from the Sequences Pane in one Sequence File Window to the Sequences Pane of another Sequence File Window. In addition, the Sequence Editor also supports dragging of sequences from the Sequences Pane and dropping them as Sequence Call steps into the Steps Pane. TestStand will create as many Sequence Call steps as the number of sequences that were dragged. These steps will be configured to call the sequences that were dropped into the Steps Pane.

You can drop sequences as steps into any sequence in the same sequence file or a different sequence file. In the Sequences View, use your mouse to drag a sequence and hover it over the sequence into which you want to drop it. When that sequence is selected, you can drop the first sequence as a Sequence Call step in the Steps Pane.

To drag and drop any sequence as a step into an already open sequence, press the <ALT> key before dragging the sequence. This action will ensure that the open

sequence remains unchanged, and allow you to choose another sequence to drag and drop into the Steps Pane.



Note You cannot use the <ALT> key shortcut to drop sequences as steps if you have configured multiple step types from which to choose. See the section below for more information about modifying the default step type.

Modifying the Default Step Type

By default, the drag and drop feature will create Sequence Call steps. You can modify the default step type by changing a configuration file setting contained in the `GeneralEngine.cfg` file located in the <TestStand Configuration> directory. The `GeneralEngine.cfg` file is saved in .xml format by default.

The `GeneralEngine.cfg` configuration file contains an XML section named: `StepTypesForGeneratingSeqCallSteps`. This section contains the following array definition, which determines the default step type:

```
<StepTypesForGeneratingSeqCallSteps classname='Strs'>
<value lbound='[0]' ubound='[0] '> <value
arrayindex='[0] '>SequenceCall</value> </value>
</StepTypesForGeneratingSeqCallSteps>
```

To add additional step types from which to choose, increase the bounds of the array definition and add new elements, as shown below:

```
<value lbound='[0]' ubound='[2] '> <value
arrayindex='[0] '>SequenceCall</value> <value
arrayindex='[1] '>MyCustomSequenceCall1</value> <value
arrayindex='[2] '>MyCustomSequenceCall2</value> </value>
```

Save `GeneralEngine.cfg` and restart the Sequence Editor for these settings to take effect.



Note If there are multiple step types specified in this configuration setting, of which more than one type can be loaded, you will be prompted to choose

among the available types when dropping a sequence into the Steps Pane. Types that cannot be loaded will be disabled in the menu.

Creating a Subsequence from Selected Steps - Sequence File Window

Creating a Subsequence from Selected Steps

You can create a new subsequence from selected steps in a sequence file. TestStand will replace the steps you select in the parent sequence with a Sequence Call step to call the new subsequence. TestStand will also create variables and update expressions for the new subsequence and will delete Local variables that are no longer referenced by the remaining steps in the parent sequence.



Note This feature is supported only in the TestStand Sequence Editor.



Note A subsequence from steps does not resolve runtime references to steps or sequences in TestStand expressions. After creation of a subsequence, review all runtime references and run Sequence Analyzer on the sequence file to ensure there are no errors.

In the Steps Pane context menu, select **New Subsequence from Steps**. You can choose to perform the operation **With Preview** or **Without Preview**. If you choose to preview the operation, TestStand will launch the Preview dialog box, which contains the following options:

- **Sequence Options**—The Preview dialog box contains the following sequence options:
 - **Subsequence Name**—Specifies the name of the subsequence. A default name is loaded when the dialog box launches.
 - **Use Current Sequence File**—This option is selected by default. When you select this option, a new subsequence will be created in the current sequence file. You can select a different sequence file in which to create the subsequence by unselecting this option.
 - **Sequence File Path**—The pathname of the sequence file in which to create the

new subsequence.

- **Delete Locals No Longer Used In Parent Sequence**—Deletes Local variables that are no longer referenced by the remaining steps in the parent sequence. This option is selected by default.
- **Subsequence Options**—The Preview dialog box contains the following subsequence options:
 - **Potential Parameters**—Lists the potential parameters that will be created in the new sequence. A variable is listed as a potential parameter if that variable is referenced by the remaining steps in the parent sequence or if that variable belongs to the Parameters variable group in the parent sequence. Unchecking **Use Current Sequence File** and selecting a different sequence file path processes the variables in the File Globals variable group in the parent sequence and adds them as parameters if a variable is used by the selected steps.
 - **Original Location**—Displays the location of the selected parameter in the parent sequence.



Note Unchecking a variable in the potential parameter list moves the variable to the original variable group (original location). If a variable's original variable group is Parameters, then unchecking is not allowed for that variable.

Expanding a potential parameter displays auto-resolved references and/or possible references:

- **Auto-Resolved References**—Lists the references that can be resolved to the selected variable at Edit time. The expressions in the referenced locations are updated to refer to the selected variable.
- **Possible References**—Lists the references that cannot be resolved to the selected variable at Edit time and can only be resolved during runtime. Expressions in the Possible References are editable.
- **Selected Reference Expression**—Displays the expression in the selected location of Auto-Resolved or Possible References.

Right-click on an auto-resolved reference or a possible reference to access the context menu. The context menu contains the following options, which are disabled by default:

- **Revert**—When you edit an expression that is listed as a possible reference, this option is enabled for that reference and any other related reference. Selecting this option reverts the expression to its original state (the state immediately following the creation of the subsequence), regardless of the number of edits you have made.
- **Redo**—This option is enabled if you choose to revert an edited expression. Selecting this option undoes the revert operation and restores the changes you made to the expression before you reverted to the original.

When you click **OK**, TestStand will create a sequence file for the new subsequence and move the steps you selected in the parent sequence into the subsequence. TestStand will also create any necessary variables and update expressions for the new subsequence. A Sequence Call step will be inserted in the parent sequence to call the new subsequence and pass appropriate parameters. If you selected the **Delete Locals No Longer Used In Parent Sequence** option, TestStand will delete Local variables in the parent sequence that are no longer referenced by the remaining steps.

Step Type Combination

Select **Combine with Step Type** from the Types window context menu to combine the functionality of two step types into a single new step type. Use this feature when the sequences commonly use steps of two different step types together to perform an operation that would be more convenient to perform with a single step. For example, you can combine a step type that reads a measurement from a specialized instrument with the Numeric Limit Test step type in order to read a measurement and check limits on it in a single step.

The Combine With Step Type option creates a new step type that has all of the custom properties and substeps both step types define. If both step types define a property with the same name or if the value of an inherent step property conflicts, the Combine With Step Type option uses the property or value for the step from which you initiate the combining operation. The new step type uses the default built-in step properties and the default module settings from the first step type.



Note The Combine With Step Type option is a step type editing tool. You must verify that the source step types do not use the same properties in conflicting ways and that the order of the substeps in the combined step type

is correct for the behavior you intend.

Importing Specifications into TestStand Using the Specifications Pane

Use the Specifications pane to connect to SystemLink Enterprise (SLE) Specification Manager or import a specification file and import data to a sequence file as variables. You can use these imported variables when configuring measurement conditions, sweeps, and test limits. If you import data from SLE Specification Manager, the SLE data plug-in also allows you to pass results tied to imported data back to SLE for analysis.



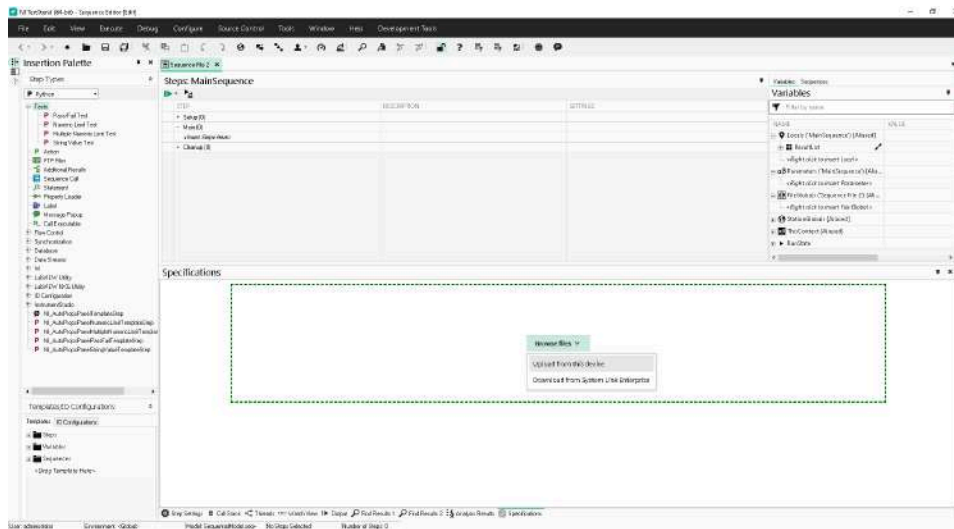
Note You can only import specifications from SLE or a specification file in 64-bit TestStand.

Importing Specifications from SLE Specification Manager

Use the Specifications pane to import product specifications from SLE Specification Manager for use in a TestStand sequence. You must define an SLE server address, API token value, and the products and categories you want to import for initial specification imports.

To import values from SLE Specification Manager into TestStand:

1. Launch the Manage Specifications dialog box by right-clicking in the Specifications pane and selecting **Browse Files » Download from SystemLink Enterprise**.



2. Enter the server information for the SLE server you want to connect to.

Manage Specifications

Connect to System Link Enterprise Specification Manager.

Server
Kindly enter System Link Enterprise URL.

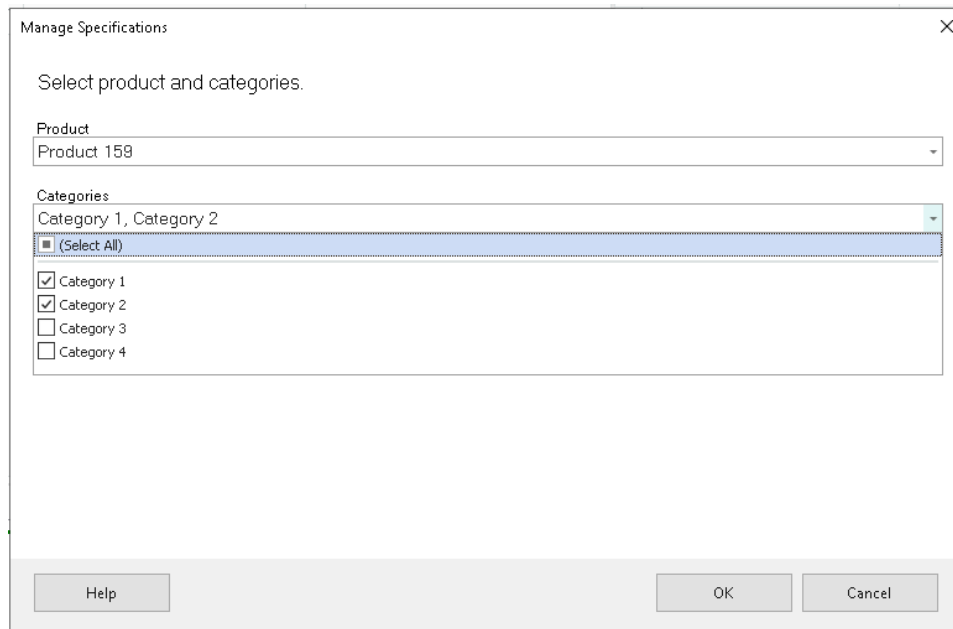
API Token

Connect

Help OK Cancel

- Enter the URL of the SLE server you want to connect to.
- Enter the API token value created in the SLE server you are connecting to.
- Click **Connect**.

3. Select the products and categories you want to import from the SLE server.



4. Click **OK** to save the product name and selected categories and load your specifications in the active sequence file.

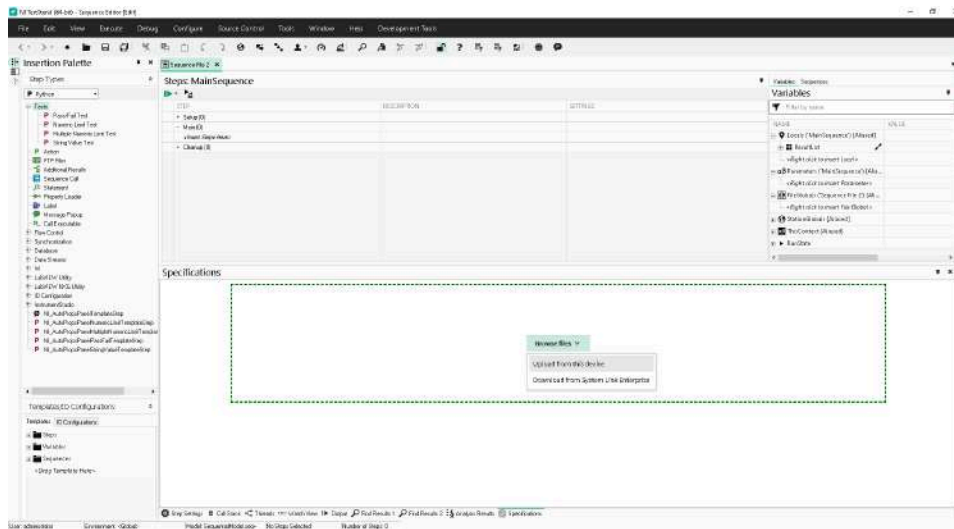
You can map imported specifications to measurement plug-ins and code modules and use them to create steps in your sequence file.

Importing Specifications from Specification Files

You can use the Specifications pane to import specification values from specification files.

You can find the template specifications file recommended for use in this process at `C:\Users\Public\Documents\National Instruments\TestStand 2024 (64-bit)\Examples\Specifications Integration`.

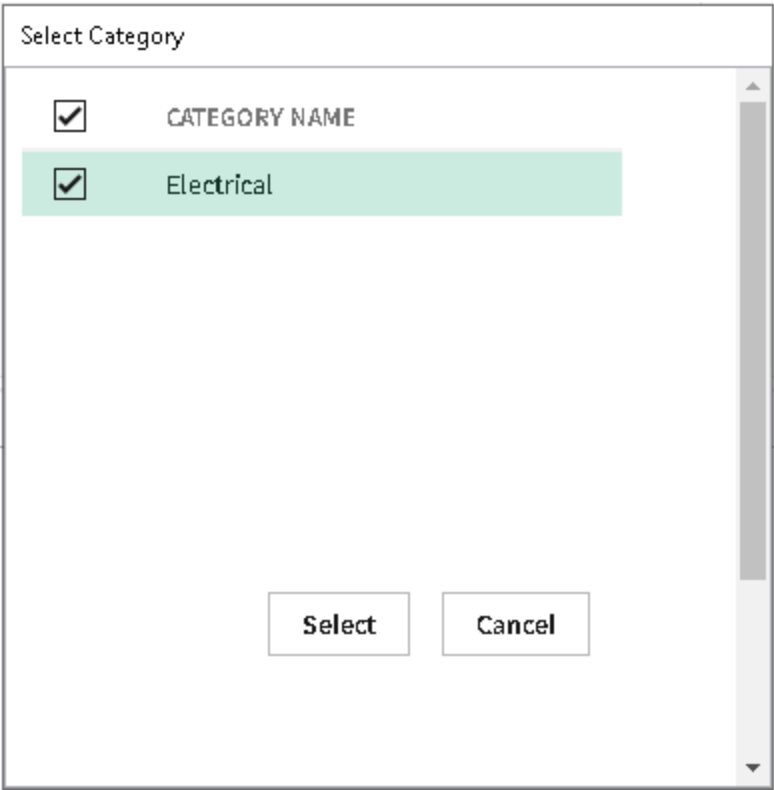
1. Launch the Manage Specifications dialog box by right-clicking in the Specifications pane and selecting **Browse Files » Upload from this Device**.



2. Select the file you want to import specifications from and click **OK**.
3. Select the specification categories you want to import. and
 - a. Open the Select Category dialog box by clicking the **Pencil** icon next to your imported specification file.



- b. Select the required categories from the category list.



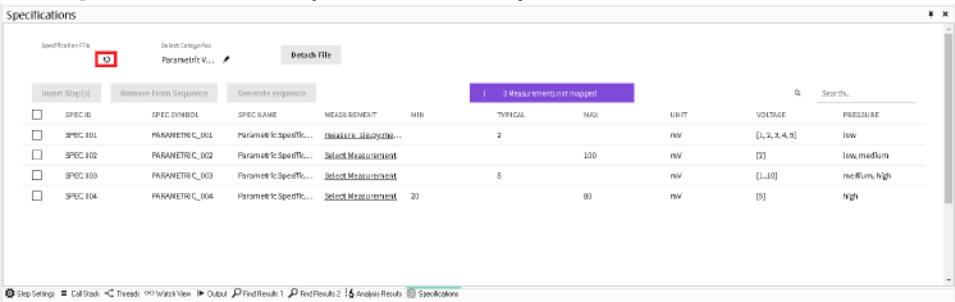
You can map imported specifications to measurement plug-ins and code modules and use them to create steps in your sequence file.

Updating Values from an SLE Server in TestStand

If product specifications imported from your SLE server or specification file change, you can update your imported specifications and any steps they impact from within TestStand.

To update specifications in TestStand with new values from an SLE server or specification file:

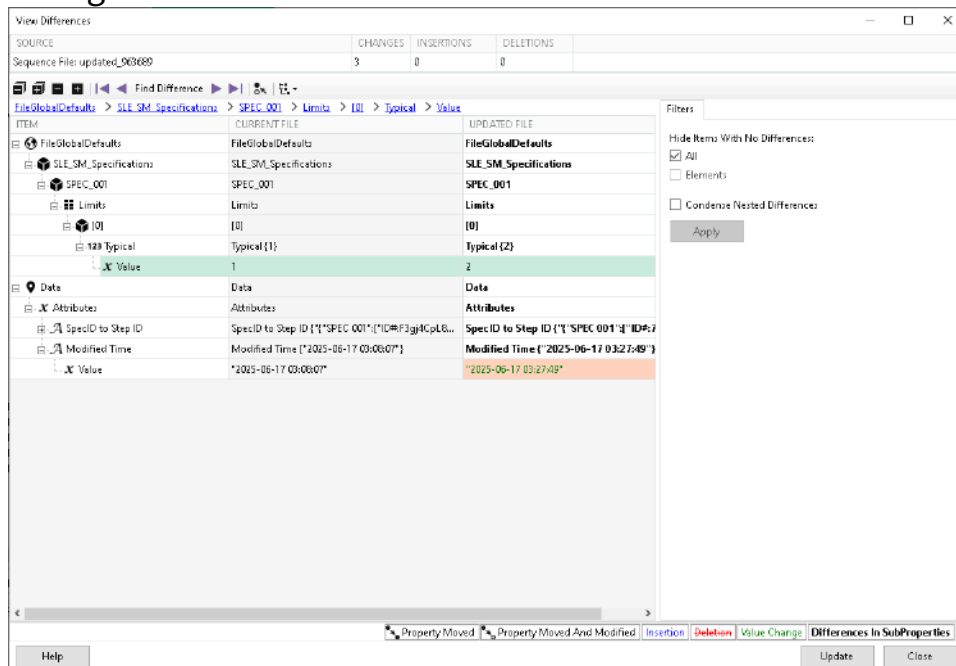
1. Retrieve the latest data from your SLE server or specification file by clicking **Check for Updates** in the Specifications pane.





Note The **Check for Updates** button is only usable if there are differences between the currently loaded specifications and the specifications in the linked SLE server or specification file.

2. Open the View Differences dialog box by clicking the **View Changes** button.
3. Preview the differences between your current and new data in the View Differences dialog box.



4. Save the updated values in the active sequence file by clicking **OK**.

Related concepts:

- [SLE Attributes, Conditions, and Specifications](#)

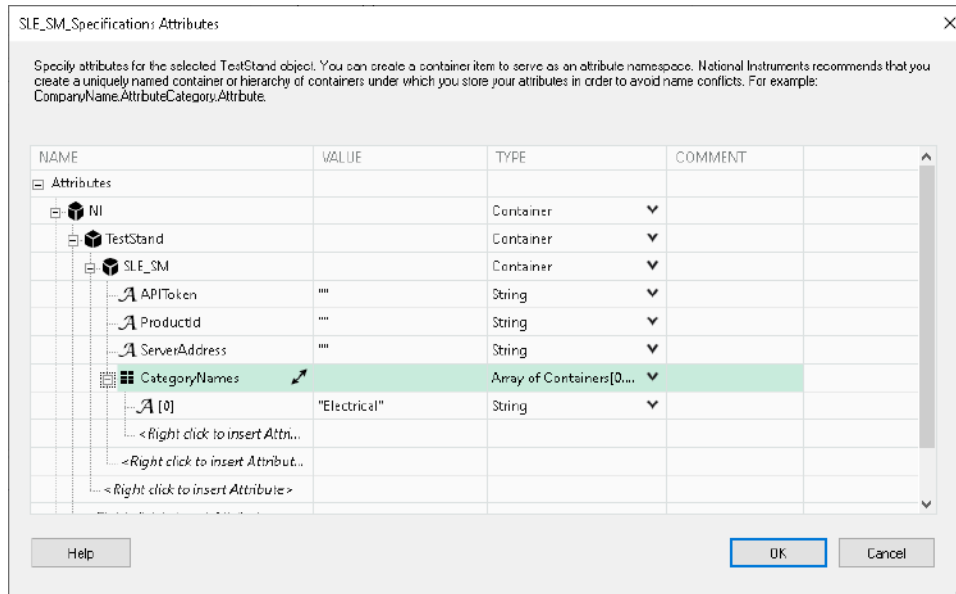
Related information:

- [Importing Values from an SCM Server into TestStand](#)

SLE Attributes, Conditions, and Specifications

Data imported into TestStand is saved in a file global for use in sequences as variables to configure measurement conditions, sweeps, and test limits.

SLE Attributes



Save SLE Attributes to the `SLE_SM_Specifications` file global by clicking **OK** in the SLE Configuration dialog box. The SLE Configuration dialog box utilizes this data to support the re-configuration of product specifications.

The saved SLE attributes are:

- **APIToken**—The SLE API Token set in the Connection Configuration view. The value is encrypted to the Windows user and machine.
- **ProductID**—The product ID set in the Product Configuration view.
- **ServerAddress**—The SLE URL set in the Connection Configuration view.
- **CategoryNames**—A list of category names set in the Product Configuration view.

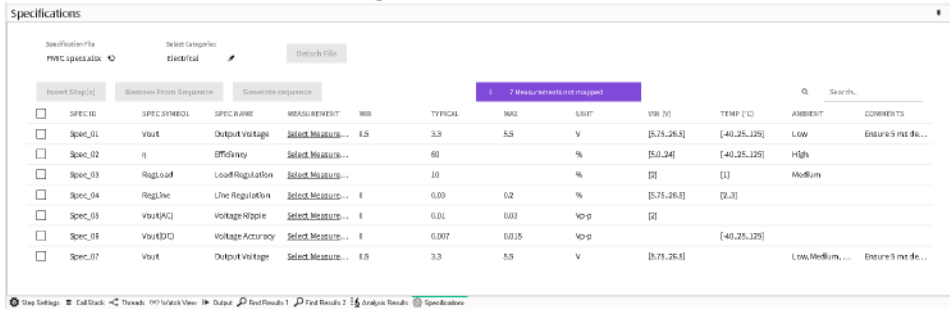
SLE Specification Conditions

The `SLE_SM_Specifications` file global contains the saved conditions for all specifications. They are the first properties listed under `SLE_SM_Specifications`. The type of each property matches the expected type of the condition in your specification source, and the property names match the names of the conditions in your specification source.

Conditions which appear like the following in a specification file:

Category Name	Electrical										
Type	Parametric										
Column Type	STD	STD	STD	STD	STD	STD	STD	STD	COND	COND	
Column Names	Spec ID	Block	Spec Symbol	Spec Name	Min	Typical	Max	Unit	V _{in} (V)	Temp (°C)	
	Spec_01		V _{out}	Output Voltage	0.5	3.3	5.5	V	[5.75..26.5]	[-40..25..125]	
	Spec_02		η	Efficiency		60		%	[5.0..24]	[-40..25..125]	
	Spec_03		Reg _{Load}	Load Regulation		0		%	[5.75..26.5]	[-40..25..125]	
	Spec_04		Reg _{Line}	Line Regulation	0	0.03	0.2	%	[5.75..26.5]	[-40..25..125]	
	Spec_05		V _{out(AC)}	Voltage Ripple	0	0.01	0.03	Vp-p	[5.75..26.5]	[-40..25..125]	
	Spec_06		V _{out(DC)}	Voltage Accuracy	0	0.007	0.015	Vp-p	[5.75..26.5]	[-40..25..125]	

appear like the following in TestStand:



SLE Specifications

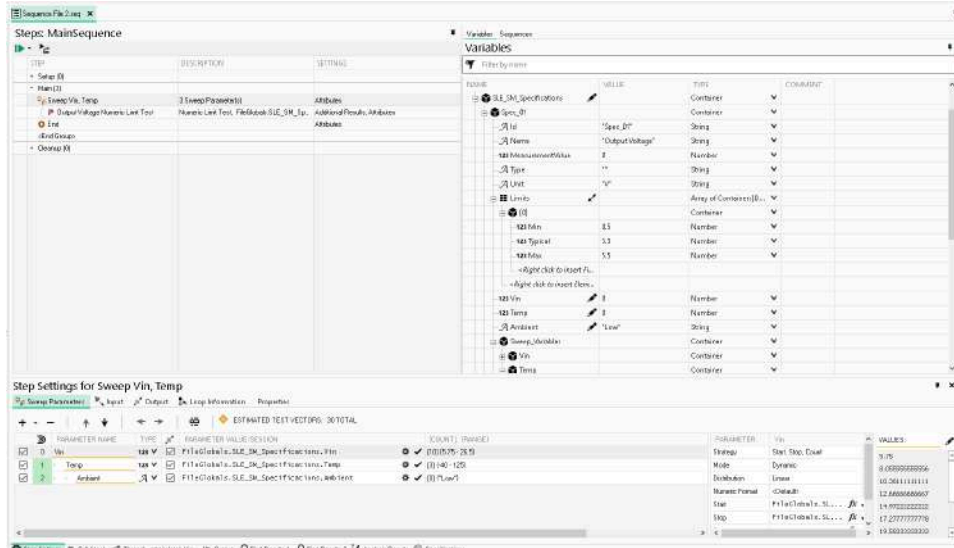
Specifications are saved as containers under the `SLE_SM_Specifications` file global. The Specifications pane uses the specification ID for the name of the container since IDs have to be unique in a product. All specification ID characters invalid in property names are replaced with underscores. Units are also ignored for property names.

Each specification container has the following properties:

- **Id**—The ID for the product specification defined by the user. This is necessary because the container name might not match the actual specification ID in SLE.
- **Name**—The descriptive name of the product specification.
- **MeasurementValue**—The current measurement for the specification. When this property is set in a sequence either through an output module parameter or an expression, it will be automatically logged along with the conditions.
- **Type**—The type of specification. Values are either `Parametric` or `Functional`.
- **<Specification Limits>**—For parametric specifications, all specification limit values are stored after the above properties. The limit property names are the following:
 - **Min**—Minimum acceptable value for the product specification during testing.
 - **Typical**—Typical value for the product specification during testing.
 - **Max**—Maximum acceptable value for the product specification during testing.
 - **Unit**—Measurement unit for the product specification.
- **<Conditions>**—The condition values for the specification are stored after the

specification limits. The names match the names in SLE when possible.

An example imported specification is shown in the following screenshot:



Condition Values

A condition value can be a number, a string, an array of numbers, an array of strings, or a numeric range.

For single values and array values, a property matching the value type is created.

For numeric ranges, properties are created based on the format of the range, as described in the following table:

Table 43. Numeric Range Formats

Numeric Range Format	Description
[MIN..]	A container is created with Min and Max sub-properties. The Max property is set to <i>INF</i> .
[..MAX]	A container is created with Min and Max sub-properties. The Min property is set to <i>INF</i> .
[MIN..MAX]	A container is created with Min and Max sub-properties.
[MIN..VAL1..MAX]	A container is created with Min, Value1, and Max sub-properties. For intermediate range values, properties with the format <i>Value<Counter></i> will be created to hold the intermediate values.

Numeric Range Format	Description
	In this example, the intermediate range value VAL1 will be stored in Value1.
[MIN..VAL1..VAL2..]	A container is created with Min, Value1, Value2, and Max sub-properties. The Max property is set to INF.

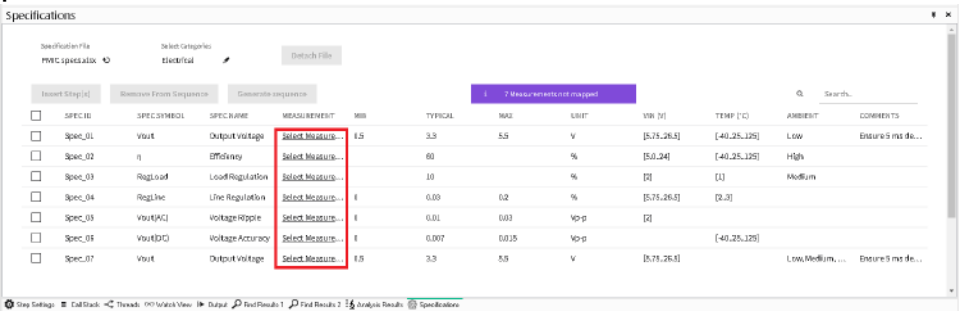
Related tasks:

- [Exporting Values to an SLE Server from TestStand](#)

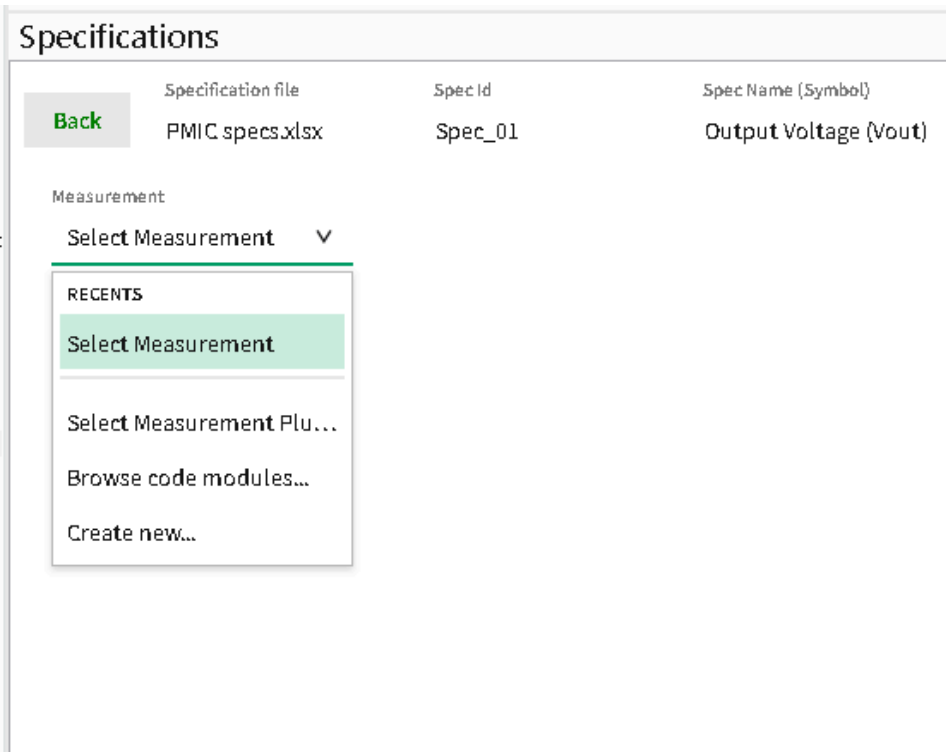
Using Imported Specifications in TestStand

Once you map the parameters of specifications you import, you can use them to create steps in your sequence file.

1. Navigate to the specification you want to map parameters for in the Specifications pane and click **Select Measurement**.



2. Use the **Measurement** drop-down list to select the appropriate measurement plugin or code module for your measurement.



Note If you do not have an appropriate code module, you can create a VI or Python code module by clicking **Create new...** in the **Measurement** drop-down list.

3. If you are mapping parameters for a measurement plug-in, map your conditions and specifications to parameters using the corresponding drop-down lists for each condition and specification.
4. If you are mapping parameters for a VI code module, complete the following steps.
 - a. If you select a LabVIEW project for your measurement, select the VI you want to map parameters for using the **Select VI** drop-down list.



Note If you select a VI directly for your measurement, you do not need to use the **Select VI** drop-down list.



Note You can only create new VI code modules through the drop-down list if you configured the LabVIEW adapter to use the LabVIEW Development Environment.

- b. Map your conditions and specifications to parameters using the corresponding drop-down lists for each condition and specification.

5. If you are mapping parameters for a Python code module, complete the following steps.
 - a. Select the function you want to map parameters for using the **Select Function** drop-down list.



Note You can only create or load Python code modules if the Python adapter is configured.

- b. Map your conditions and specifications to parameters using the corresponding drop-down lists for each condition and specification.
6. Once you have mapped your parameters, click **Back**, select your mapped specification from the list in the Specifications pane, and click **Insert Step** to create a step at the end of `MainSequence` your sequence file for obtaining the selected specification.



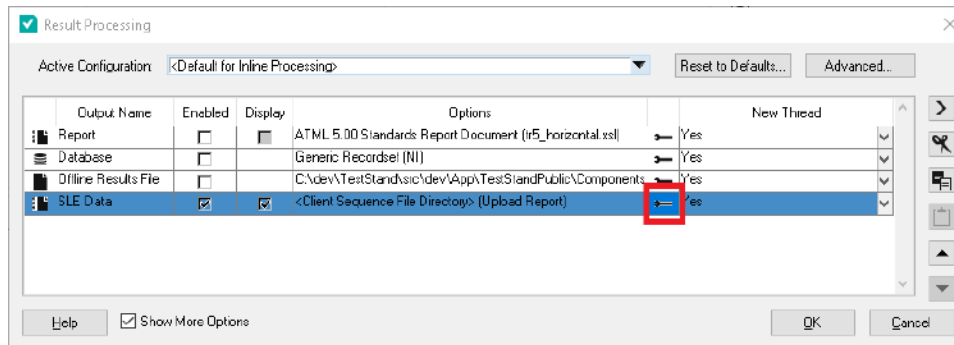
Note If you do not create a step using **Insert Step** and your imported specifications, TestStand will not save your imported specifications and parameter mappings.

Exporting Values to an SLE Server from TestStand

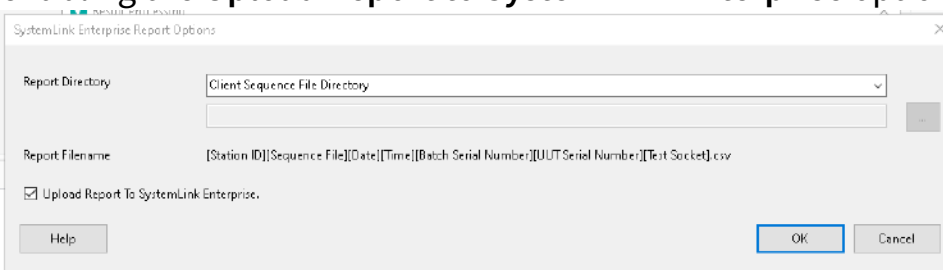
When specifications are imported into a sequence file from an SLE server, the SLE data result processing plug-in is automatically added to the active processing configuration. This plug-in generates a measurement data file with all measurement values logged during an execution. You can upload these measurement values to SLE at the end of an execution.

To export specifications from TestStand to an SLE server, Specify options for the generated SLE data report file using the SystemLink Enterprise Report Options dialog box:

1. Open the SystemLink Enterprise Report Options dialog box by clicking on the icon in the **Options** column for the SLE data reporting plug-in in the Result Processing dialog box.



2. Set the directory where the report generator writes the report file using the **Report Directory** ring control.
3. Ensure you upload the SLE data report file to SLE after the report is generated by enabling the **Upload Report to SystemLink Enterprise** option.



Note The **Upload Report to SystemLink Enterprise** option is enabled by default, so uploading the measurement data file to SLE is the default behavior of the SLE data result processing plug-in.

TestStand Deployment Utility

You can launch the TestStand Deployment Utility in the following ways:

- Select **Tools»Deploy TestStand System** in the TestStand Sequence Editor.
- (Windows 8.1) Click the **NI Launcher** tile on the Start screen and select **TestStand»Tools»TestStand Deployment Utility**.
- (Windows 10 or 7) Select **Start»All Programs»National Instruments»TestStand»Tools»TestStand Deployment Utility**.

Use the deployment utility to create a deployable image or a patch deployment of a TestStand system and an optional installer. The deployable image can contain sequence files, code modules, process models and supporting files, user interface applications, configuration files, and step types and supporting files the TestStand system uses. The installer can contain all files from a deployable image or contain only

a subset of files to create a patch deployment for a previously deployed image.



Note You must save the TestStand deployment specification file (.tsd) and the full deployment installer to create a patch deployment and installer.

You can also use the deployment utility to include the TestStand Runtime and supporting files, LabVIEW and LabWindows/CVI Run-Time Engines, and hardware drivers in the installer you create. If the deployment includes LabVIEW files, you must have the LabVIEW development system with the Application Builder installed to automatically detect dependencies of top-level VIs and create a deployable image or installer. You do not need the Application Builder to use the deployment utility if you do not deploy LabVIEW files or if you enable the Include without Processing Item or Dependencies option on the Distributed Files tab of the deployment utility. You can also configure the deployment utility to remove VI block diagrams or to lock the VIs you deploy. The installer that the deployment utility creates can also register ActiveX servers, replace existing files on the target computer, and create program shortcuts.



Note For each system that you deploy, you must activate a TestStand license, typically a unique copy of the TestStand Base Deployment System License or the TestStand Debug Deployment Environment License.



Note The TestStand Deployment Utility does not require a license for build or analysis purposes, as defined by the License Agreement.

The deployment utility window contains the following tabs:

- **Mode**—Use this tab to create a full deployment or a patch deployment and to specify related settings for each type of deployment.
- **System Source**—Use this tab to specify the workspace file or directory that specifies which files to deploy and the location to create the deployment image.
- **Distributed Files**—Use this tab to view the files included in the deployment and configures the installer destination location and settings for each file.
- **Installer Options**—Use this tab to configure installer settings for the deployment and specifies which components to include from TestStand and other National Instruments software, such as hardware drivers.
- **Package Installer Options**—Use this tab to configure package-based distribution settings.

- **Build Status**—Use this tab to view progress and log information when analyzing files to include in a deployment and when building a deployment image and installer.

The deployment utility also contains the following buttons:

- **Load**—Loads the deployment settings from a previously saved (.tsd) file.
- **Save**—Saves the current deployment settings to the currently loaded (.tsd) file. When you create a patch deployment, the deployment utility appends the patch deployment information to the .tsd file.
- **Save As**—Saves the current deployment settings to a new (.tsd) file.
- **New**—Resets the current deployment settings to a new (.tsd) file.
- **Build**—Creates a deployable image and installer, according to the settings on the Installer Options tab.
- **Done**—Exits the deployment utility.

Related concepts:

- [Patching Deployments](#)
- [Sequence Files](#)
- [Code Modules](#)
- [Process Models](#)
- [Deploying the TestStand Runtime](#)
- [Activating and Licensing TestStand](#)
- [Deploying TestStand Systems](#)

Related information:

- [TestStand Deployment Utility Environment Reference](#)
- [TestStand Deployment Utility Messages](#)

TestStand File Diff and Merge Utility

Use the TestStand File Diff and Merge utility to compare and merge non-type differences among two or three sequence files and compare type differences among two sequence files or type palette files. You can launch this utility in the following ways:

- (Windows 8.1) Click the **NI Launcher** tile on the Start screen and select

TestStand»Tools»TestStand File Diff and Merge Utility.

- (Windows 10 or 7) Select **Start»All Programs»National Instruments»TestStand Shared»TestStand File Diff and Merge Utility**.
- Run `<TestStand>\Bin\FileDiffer.exe`.

In the Sequence Editor, you can select the **Edit»Diff Sequence File Against** and **Edit»Diff Against Saved Version** menu options to launch the TestStand File Diff and Merge Utility to compare an opened sequence file to a file on disk.

The Select Files dialog box launches when you launch the utility. Use the Diff tab of the Select Files dialog box to specify two sequence files or type palette files you want to compare. Use the Merge tab to specify three sequence files you want to compare—a base file from which modified versions originate and two modified versions—and a fourth sequence file to become the resulting merged file to create.

Click the **Options** button in the Select Files dialog box to launch the File Differ Options dialog box, in which you can configure settings for the utility, including options for loading and automatically merging types, managing type conflict resolution, and so on.

When you specify files to compare in the Select Files dialog box and click **OK**, the utility launches the main window, which contains the following information:

- **File List**—The specified files you are comparing and a count of changes, insertions, and deletions for each file.
- **Differences Table**—Displays the differences grouped by sequences, file global variables, file properties, and types for each specified file.
- **Filters Tab**—Hide properties with no differences in the Differences window.

Source Code Control

TestStand includes a shared launching application you can use with third-party source code control (SCC) providers and Git clients that support launching an external application to perform two- and three-way file diff and merge operations. The launching application determines the active version of TestStand and launches the corresponding TestStand diff application that version of TestStand installs. Using the launching application alleviates the requirement to reconfigure the SCC provider or Git client each time you activate a different version of TestStand. Launch the application

by running `<National Instruments>\Shared\TestStand\FileDifferLauncher.exe`.

Command-Line Usage

You can also specify files to compare using command-line arguments for the application. You typically use the application to compare files in a source code control system. For example, use the following command-line syntax to specify file paths to a base file, two modified versions, and the resulting merged file to create:

```
FileDiffer.exe "C:\My Documents\File1.seq"
"C:\My Documents\File2.seq" "C:\My Documents\File3.seq"
"C:\My Documents\Merged.seq"
```

Use the following command-line syntax to launch a dialog box that lists all the command-line arguments you can use for the application.

```
FileDiffer.exe /?
```

or

```
FileDiffer.exe /help
```

To launch the File Diff and Merge Utility in a specific TestStand environment, use the following command line argument:

```
/env <environment path>
```

where `<environment path>` is the location of the `.tsenv` file that defines the environment.



Note The TestStand File Diff and Merge Utility does not require a license for build or analysis purposes, as defined by the License Agreement. If you do not have a license, you can diff, but not save, sequence files.

The following command-line tokens are supported in the help.

- `/env <environment path>`—Specify the TestStand environment.
- `/matchStepsByID: {Y|N}`—Require step IDs to match. Provide N for disabling the option.
- `/TypeMerging: {Y|N}`—Enable or disable automatic type merge.
- `/ProcessModelLoading: {Y|N}`—Enable or disable process model loading.
- `/TypePaletteLoading: {Y|N}`—Enable or disable loading type palettes.
- `/IgnoreShared: {Y|N}`—Enable or disable ignoring differences in variables with the shared property.
- `/AutoResolveShared: {N|Base|File1|File2}`—Automatically resolve ignored shared conflicts. Requires `/IgnoreShared:Y`.
- `/IgnoredMinorModule: {Y|N}`—Enable or disable ignoring minor module differences.
- `/AutoResolveMinorModule: {N|Base|File1|File2}`—Automatically resolve ignored minor module conflicts. Requires `/IgnoreMinorModule: {Base|File1|File2}`.
- `/GenerateReport <reportPath.xml>`—Automatically generate the diff report and then exit. To have this run synchronously from a batch script use the 'start' command with the '/wait' parameter to launch the process.

Related concepts:

- [Selecting a Source Code Control Provider](#)
- [Search Directories](#)
- [TestStand Environments](#)

Related information:

- [Git Pane](#)
- [TestStand File Diff and Merge Utility Environment Reference](#)

TestStand Offline Results Processing Utility

You can launch the TestStand Offline Results Processing Utility in the following ways:

- Select **Tools»Offline Results Processing Utility** in the TestStand Sequence Editor.
- (Windows 8.1) Click the **NI Launcher** tile on the Start screen and select **TestStand»Tools»Offline Results Processing Utility**.

- (Windows 10 or 7) Select **Start»All Programs»National Instruments»TestStand»Tools»TestStand Offline Results Processing Utility**.

Use this utility to process raw results files into reports and database tables at a time other than when sequence execution occurs, such as overnight, or on a different computer. Use this utility in situations in which performance is more important than immediately processing results. You can use 64-bit TestStand to process files that 32-bit TestStand generates and vice versa. If 32-bit TestStand returns out of memory errors while processing a TSR file, process the file with 64-bit TestStand, which can use more memory during processing.

Select **Profile»New** or right-click in the Profiles pane and select **New Profile** from the context menu to create a profile. Use profiles to define a set of raw results files to process, to specify the result processing configuration set to use to process results, to specify the sequence file to use as the callback sequence file when processing the raw results files, and to track the set of files the utility has already processed. Use the inbox of a profile to specify where to store raw results files to process, and use the outbox of a profile to specify where to store raw results files and reports after processing. You can configure the number of files the utility processes simultaneously.

The utility creates a new execution to process raw results files into reports and database tables. This execution calls callbacks for offline results processing in the same manner as normal results processing. Use the **Default Callback File** option in the TestStand Offline Results Processing Utility Settings dialog box to specify a default callback file for all profiles, or specify a callback file for each profile in the profile grid.

The utility selects the callback file used to process results based on the following criteria:

1. If you specify a callback file for a profile, the utility will use that callback file to process results.
2. If you do not specify a callback file for a profile, the utility will use the default callback file to process results.
3. If you do not specify a callback file for a profile and do not specify a default callback file, the utility will not use any callback file.

When files exist in the inbox of a profile, the utility will load both the specified callback file for the profile and the default callback file, even when the profile is paused. The

utility will unload the callback files shortly after the last file is moved out of the inbox.

Which callbacks are called during offline results processing depends on the plug-ins used to deserialize. The same callbacks would be called during normal results processing for the same plug-ins.



Note When the utility generates a report after processing, it adds a TestStand Results (TSR) ID to each report file name. The TSR ID is a unique string identifying the raw results file from which the report was generated. After processing a raw results file, the utility automatically moves the file to the outbox. If another raw results file with the same name exists in the outbox, the utility automatically appends the TSR ID to the end of the name of the processed file.

The Report List pane displays the list of reports the utility generated from the raw results file of the same name. Use the **Next Report** and **Previous Report** buttons to show the next or previous unread report. An unread report is one that the utility has not opened during the current session. This pane also uses the same controls as the Report pane in the sequence editor Execution window.

Select **File»Settings** to launch the TestStand Offline Results Processing Utility Settings dialog box, in which you set utility configuration options. If you are processing TSR files with a large number of results, use the **Number of Files Processed Simultaneously** option in this dialog box to reduce memory usage.

The status bar of the utility displays the default callback file and the report you selected in the Report List pane.

You can use command-line arguments or a Call Executable step with command-line arguments in a sequence to launch the offline results processing utility.

You can also use scheduled tasks in Microsoft Windows to launch the offline results processing utility with command-line arguments. The utility is located at `<TestStand>\Components\Tools\Offline Results Processing Utility\OfflineResultsProcessingUtility.exe`. If you want the offline results processing utility to run a specified amount of time, create two tasks and specify the `/quit` argument for the second task to stop the utility immediately. Refer

to Microsoft documentation for more information about scheduling tasks.



Note National Instruments provides the utility source code only for informational purposes and does not recommend rebuilding the utility from the source code.



Note The Offline Results Processing Utility does not require a license for build or analysis purposes, as defined by the License Agreement.

Related concepts:

- [Search Directories](#)

Related information:

- [Call Executable Step](#)
- [Execution Window](#)
- [TestStand Offline Results Processing Utility Environment Reference](#)

Logging Offline Results Processing Utility Messages to a File

Run the TestStand Offline Results Processing Utility with a redirection operator to store a log file of the information the utility writes to the command-line standard output. Refer to Microsoft documentation for more information about using command redirection operators.

For example, use the following command to append all information or higher level messages to the end of a file named logfile.txt:

```
OfflineResultsProcessingUtility.exe /log-level Info >>  
logfile.txt
```

Use the following command to log all error or higher level messages to a file named logfile.txt, overwriting the file if it already exists:

```
OfflineResultsProcessingUtility.exe /log-level Error >  
logfile.txt
```

XML Packaging Utility

Use the XML Packaging Utility to copy local XML and HTML report files you select, including related style sheets, image files, and referenced files, to a destination directory you specify so you can view the XML report on a computer that does not have TestStand installed. You can launch the utility in the following ways:

- Select **Tools»Package XML/HTML Files for Distribution** in the TestStand Sequence Editor.
- (Windows 8.1) Click the **NI Launcher** tile on the Start screen and select **TestStand»Tools»TestStand XML Packaging Utility**.
- (Windows 10 or 7) Select **Start»All Programs»National Instruments»TestStand»Tools»TestStand XML Packaging Utility**.

When you use the XML Packaging Utility to distribute XML, HTML, and supported versions of ATML reports that include hyperlinks, the utility also distributes the target file of the link if the `XmlPack.ini` file includes the file extension. You must run the utility at least once for TestStand to create the `XmlPack.ini` file, which is located in the `<TestStand Application Data>\Cfg` directory. Edit the `COMMON_EXTENSIONS` key of the `XmlPack.ini` file to include a comma-separated string of file extensions for files you want to distribute using the utility. For example, if you want to distribute `.docx` and `.xlsx` files, use `"docx, xlsx"` as the `COMMON_EXTENSIONS` key. TestStand distributes `.tdms` files by default.



Note The utility packages only local files and not remote files that are referenced by the report files.

After the utility completes the packing, the selected and referenced files reside in the specified destination directory. The utility substitutes all file references to use the files copied to the destination directory. For distribution, you can compact, deliver, and extract the destination directory to view the copied files at the distribution location.

To package XML files, first click **Add** in the XML Packaging Utility dialog box and select one or more XML files to package. The files appear in the XML Files list. Use the **Destination Folder** control to specify the directory in which the utility copies the files to in the Destination Folder control. Click **Pack** to package the XML and other referenced files.

The XML Packaging Utility dialog box contains the following options:

- **XML Files**—The XML files to be processed.
- **Add**—Launches the Add XML files dialog box, in which you can specify the XML files to add to the XML Files list.
- **Remove**—Deletes the file or files selected in the XML Files list.
- **Pack**—Packages the files in the XML Files list and copies them to the directory the Destination Folder control specifies. When packaging completes, the Referenced Files list shows the packaged files. The utility stops packaging files if an error occurs during packaging.
- **Destination Folder**—The directory in which the utility copies the packaged files. You can specify an existing directory or enter the path of a new directory for the utility to create. If the specified directory does not yet exist and you click **Pack**, the utility prompts you to confirm that you want to create it. If you do not confirm the creation of the new directory, the utility stops packaging the files.
- **Referenced Files (packaged)**—After the utility completes the packaging process, this control lists the files the utility copies to the directory the Destination Folder control specifies. The utility renames any duplicate filenames using an incremental suffix and updates any reference to a duplicated name. If the packaging process fails, or if an XML file is added or removed, the list of packaged files is reset.
- **Referenced Objects (not packaged)**—After the utility completes the packaging process, this control lists the objects the packaged files reference, such as ActiveX or Java class objects. The utility does not package these objects with the files but lists them in this control so you can verify that the objects exist in the destination directory. If an XML file is added or removed, the list of referenced objects is reset.
- **Exit**—Closes the XML Packaging Utility dialog box.

When packaging the files, the utility displays the following icons to the left of the XML Files list and the Referenced Files list:

- **Checkmark**—Indicates that the utility successfully parsed an XML file.
- **X**—Indicates that an error occurred for the file.
- **Diamond**—Indicates that an HTML file is not in XHTML format but the utility processed the file.

If no icon displays, the utility copied the file but did not process it.



Note The XML Packaging Utility does not package the TsGraphControl2.ocx ActiveX control, located in the <National Instruments>\Shared\TestStand directory.



Note The XML Packaging Utility does not require a license for build or analysis purposes, as defined by the License Agreement.

Command-Line Usage

You can execute the utility from the command line using the following syntax:

```
XMLPack [[-v] file-list [ file-list* ] dest-folder]
```

The following table describes the elements of this command-line syntax.

Syntax Element	Description
-v	Represents verbose mode, which displays messages.
file-list	A valid directory or path and filename that can contain wildcard characters, such as * and ?.
dest-folder	A valid directory or path, which might not exist. If it does not exist, TestStand creates it.

Square brackets contain optional command-line arguments. If you do not specify an argument, the dialog-based XML Packaging Utility launches.

When run from the command line, the utility returns an error code of 0 if successful and non-zero if an error occurred.

Related concepts:

- [Including Hyperlinks in a Report](#)
- [Search Directories](#)
- [NI TDMSReference](#)
- [Displaying Measurement Data as Graphs](#)

Related information:

- [XML Packaging Utility Environment Reference](#)

TestStand Sequence Analyzer

Use the TestStand Sequence Analyzer in the TestStand Sequence Editor or the stand-alone sequence analyzer application to complete the following tasks for workspace files, sequence files, directories, type palette files, station globals files, template files, and users files during development or before deployment:

- Find errors
- Enforce custom development guidelines you establish
- Gather statistics

The sequence analyzer uses a built-in set of rules and analysis modules to analyze the files you specify and generate messages that correspond to each issue found during analysis. The built-in rules are designed to detect, at edit time, the most common situations that can cause run-time failures.

Use the Analysis Results pane in the sequence editor or the Messages tab in the sequence analyzer application to view and resolve the messages for the most recent analysis of the current analyzer project, sequence file, or workspace file.

You can create custom rules and analysis modules for the sequence analyzer to use. Refer to the files located in the `<TestStand Public>\Examples\TestStand Debugging Features\Custom Analyzer Rules` directory for examples of custom rules and analysis modules. You can also distribute custom rules and related supporting files to use in analysis projects or to analyze files on other computers on which TestStand is installed.

In the TestStand Sequence Editor, click the **Current Sequence Analyzer Project** button on the Sequence Analyzer toolbar or select **Debug»Sequence Analyzer»Current Sequence Analyzer Project** to open the Current Sequence Analyzer Project window and the most recently opened analyzer project file.

Related concepts:

- [Search Directories](#)
- [Creating Analysis Modules for Custom Rules](#)
- [TestStand Sequence Analyzer Rules Descriptions](#)
- [Custom Analyzer Rules](#)

Related tasks:

- [Creating Custom Rules in the TestStand Sequence Analyzer](#)

Related information:

- [Current Sequence Analyzer Project Window](#)
- [TestStand Sequence Analyzer Environment Reference](#)
- [TestStand Sequence Analyzer API Reference](#)

TestStand Sequence Analyzer Application

Use the TestStand Sequence Analyzer application to analyze projects outside of the TestStand Sequence Editor. You can launch the application in the following ways:

- (Windows 8.1) Click the **NI Launcher** tile on the Start screen and select **TestStand»Tools»TestStand Sequence Analyzer**.
- (Windows 10 or 7) Select **Start»All Programs»National Instruments»TestStand»Tools»TestStand Sequence Analyzer**.
- Run `<TestStand>\Bin\AnalyzerApp.exe`.

The sequence analyzer application window contains the following tabs:

- **Files**—The files and directories to analyze.
- **Rules**—The rules to use for analysis. Use this tab to enable, disable, and configure rules for analysis in the current project.
- **Options**—Contains the project options that control the analysis and determine whether the sequence editor automatically saves the current analyzer project.
- **Messages**—Contains the list of messages for the most recent analysis of the current project, sequence file, or workspace file. The sequence analyzer overwrites the content of the Messages tab each time you start an analysis session.
- **Analysis Summary**—Contains a summary of the most recent analysis of the project.

Command-Line Usage

Refer to the following table for more information about the commands you can perform in the sequence analyzer application. Commands are listed in order of execution.

Order	Command	Behavior
1	<code>/minimize</code>	Launches the interface in a minimized state.
2	<code>/analyze</code>	Analyzes the open project.
3	<code>/report</code>	Generates a report for the most recently performed analysis. If no analysis has been performed, a dummy report will be generated. If the <code>/analyze</code> command is present, the report will not be generated until the analysis completes. If you specify a file path after the <code>/report</code> command, the report will be placed in that location. If you specify a relative file path, the path will be assumed relative to the open analyzer project directory. If you do not specify a file path, the report will be placed in the analyzer project directory.
4	<code>/save</code>	Saves the open project. If the <code>/report</code> command is present, the project will not save until the report has been generated.
5	<code>/clearMessagesOnSave</code>	Deletes analysis messages from the open project on save.
6	<code>/quit</code>	Closes the interface and quits all associated processes. If the <code>/analyze</code> , <code>/report</code> , or <code>/save</code> commands are present, the application will not quit until these operations are complete.
—	<code>/env <environment path></code>	Launches the application in the environment defined in the <code>.tsenv</code> file located at <code><environment path></code> .
—	<code>/help</code>	Prints information about command-line syntax usage to the command line.

You can use the following command-line syntax to open an analyzer project:

```
AnalyzerApp.exe "C:\My Documents\MyProject.tsaproj"
```

Use the following command-line syntax to perform commands on an open analyzer project:

```
AnalyzerApp.exe "C:\My Documents\MyProject.tsaproj"
/analyze /report /save /clearMessagesOnSave
```

Sequence Analyzer ExitCodes

The Sequence Analyzer application returns a negative value if there is a problem running the Sequence Analyzer, and returns a positive value if analysis reports one or more rule violations. The following are valid return codes, listed in descending order of precedence:

Number	Result	Behavior
-2	Invalid Paths	An invalid path was passed.
-1	Invalid Flags	An invalid flag was passed.
1	Analyzer Error	One or more errors were reported when analyzing the project.
2	Analyzer Warning	One or more warnings were reported when analyzing the project.
0	Success	No Errors or Warnings were reported.

Note: If both warnings and errors are reported during the analysis of a project, the return value indicates an error.



Note The Sequence Analyzer application does not require a license for build or analysis purposes, as defined by the License Agreement.

Related concepts:

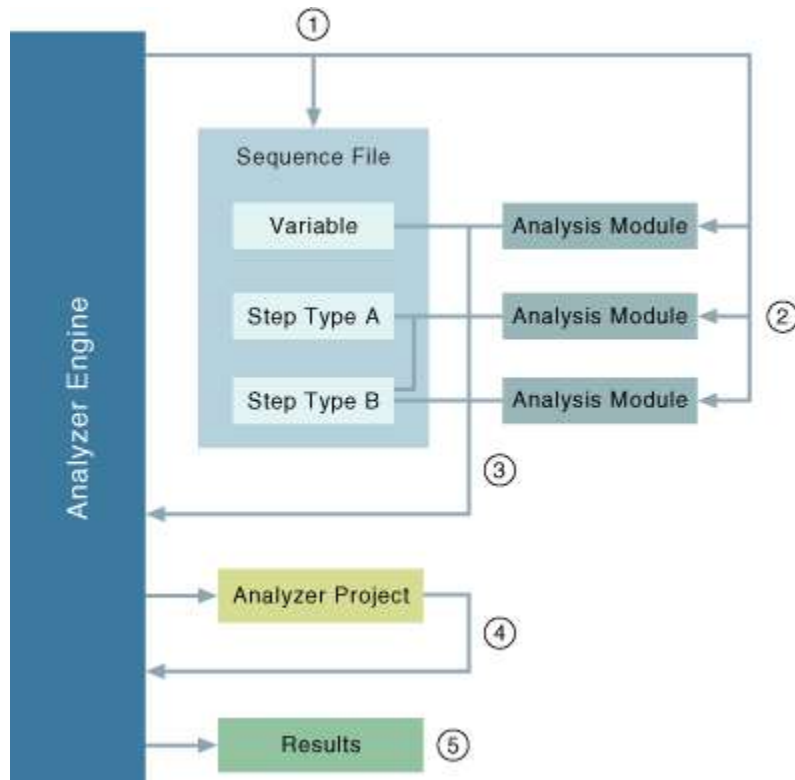
- [Search Directories](#)
- [Custom Analyzer Rules](#)

Related information:

- [TestStand Sequence Analyzer Environment Reference](#)

Analyzing Files

The TestStand Sequence Analyzer completes the tasks described in the following figure to analyze workspace files, sequence files, directories, type palette files, station globals files, template files, and users files.



1. The analyzer engine begins analyzing the workspace file, sequence file, directory, type palette file, station globals file, template file, or users file.
2. The analyzer engine checks for analysis modules that apply to the analyzed file. For example, if an analysis module applies to a variable, the analyzer invokes the code module the analysis module specifies for each variable in the analyzed file. An analysis module can apply to more than one element of a file, or you can apply more than one module to one element of a file.
3. Each code module an analysis module calls reports any broken rules to the analyzer engine.
4. The analyzer engine filters the list of reported rules through the sequence analyzer project, which specifies whether some rules are disabled, the severity of each enabled rule, and the description to display for each enabled rule.
5. The analyzer engine outputs the severity, description, and location for each triggered, enabled rule to the Analysis Results pane in the sequence editor or the

Messages tab in the stand-alone sequence analyzer application.

Related concepts:

- [Code Modules](#)
- [Creating Analysis Modules for Custom Rules](#)
- [TestStand Sequence Analyzer Rules Descriptions](#)

Related tasks:

- [Creating Custom Rules in the TestStand Sequence Analyzer](#)

Analyzing a Sequence File or Workspace File in the TestStand Sequence Editor

Complete the following steps to use the TestStand Sequence Analyzer to analyze a sequence file or workspace file.

1. In the TestStand Sequence Editor, click the **Current Sequence Analyzer Project** button on the Sequence Analyzer toolbar or select **Debug»Sequence Analyzer»Current Sequence Analyzer Project** to open the Current Sequence Analyzer Project window and the most recently opened analyzer project file.



Note The first time you launch Current Sequence Analyzer Project window, the sequence editor creates a default project, named `MyAnalyzerProject.tsaproj`, which specifies the rules and related settings to use during the analysis. You can create, modify, and save project files to customize the set of rules, rule configuration settings, and set of files to use during analysis.

2. On the Rules pane, enable and configure the rules you want to use to analyze the file.



Note The sequence analyzer enables most of the built-in rules by default. You cannot disable or delete the built-in rules in the Analysis Errors category.

3. Open and activate the sequence file or workspace file you want to analyze. Click

the **Analyze <Sequence Filename>** button on the Sequence Analyzer toolbar to analyze a sequence file. Right-click the workspace file and select **Analyze Workspace** from the context menu to analyze a workspace file.



Note By default, the sequence analyzer enables the Analyze File Before Executing option in the Sequence Analyzer Options dialog box. In the sequence editor, click the **Toggle Analyze File Before Executing** button on the Sequence Analyzer toolbar or select **Debug»Sequence Analyzer»Toggle Analyze File Before Executing** to enable or disable this option. When you enable this option, the sequence analyzer uses the rules and settings in the current sequence analyzer project to analyze the active sequence file when you initiate an execution. Enabling this option can help you find errors early during development.

4. The status bar at the bottom of the Analysis Results pane indicates the progress of the analysis. You can click the **Stop Analysis** button on the Sequence Analyzer toolbar or select **Debug»Sequence Analyzer»Stop Analysis** at any time during analysis. The sequence analyzer finishes analyzing the current object before stopping the analysis and shows the current list of messages on the Analysis Results pane.
5. (Optional) When you complete an analysis, you can save an XML report file to use in an external viewer for comparison. In the sequence editor, click **Generate Analysis Results** on the Analysis Results pane to save the messages for the most recent analysis in an XML report file to use in an external viewer. The XML file uses the AnalyzerReportViewer.xsl style sheet, located in the `<TestStand>\Components\Stylesheets\Analyzer` directory. When you generate a report file, the sequence analyzer provides a snapshot of the most recent analysis, including a list of the analyzed files, a list of the rules and settings used during the analysis, and a list of all the generated messages.



Note Use the XML Packaging Utility to package the files necessary to view the XML report on a computer that does not have TestStand installed.

6. Use the **Show** and **Group By** drop-down lists to control how to view the generated messages on the Analysis Results pane. Right-click each message and select an option from the Analyzer Messages tab context menu to resolve the message.

For each active message, you can perform the following actions:

- a. Go to the location of the object in the TestStand file that caused the message and resolve the issue.
- b. Ignore the issue for subsequent analysis and remove it from the Active Messages view. Select this option only when you are certain the issue will not cause a run-time error.
- c. Mark the message as fixed and remove it from the Active Messages view.
- d. Disable or configure the corresponding rule on the Rules pane of the sequence analyzer.



Note You can save the project file at any time when you are resolving messages.

You can repeat these steps to cycle through the analysis, report generation, and message resolution stages until you attain acceptable results.

Related concepts:

- [Search Directories](#)
- [TestStand Sequence Analyzer Rules Descriptions](#)
- [XML Packaging Utility](#)

Related tasks:

- [Creating Custom Rules in the TestStand Sequence Analyzer](#)

Related information:

- [Analysis Results Pane](#)

Analyzing Multiple TestStand Files Using the TestStand Sequence Analyzer

Complete the following steps to use the TestStand Sequence Analyzer to analyze multiple TestStand files.

1. In the TestStand Sequence Editor, click the **Current Sequence Analyzer Project** button on the Sequence Analyzer toolbar or select **Debug»Sequence Analyzer»Current Sequence Analyzer Project** to open the Current Sequence

Analyzer Project window and the most recently opened analyzer project file.



Note The first time you launch the Current Sequence Analyzer Project window, the sequence editor creates a default project, named `MyAnalyzerProject.tsaproj`, which specifies the rules and related settings to use during the analysis. You can create, modify, and save project files to customize the set of rules, rule configuration settings, and set of files to use during analysis.



Note If you want to use the stand-alone sequence analyzer application, select **Start»National Instruments»TestStand»Tools»TestStand Sequence Analyzer** to launch the application. Click the **New Sequence Analyzer Project** button on the Sequence Analyzer toolbar or select **File»New Sequence Analyzer Project** to create a new project.

2. On the Files pane or tab, select the files or directories you want to analyze. When you select a workspace file, the sequence analyzer automatically analyzes all the sequence files in that workspace file.
3. On the Rules pane or tab, enable and configure the rules you want to use in the current project.



Note You cannot disable or delete built-in rules in the Analysis Errors category.

4. On the Options pane or tab, enable or disable the analyzer project options you need.
5. Click the **Analyze <project filename>** button on the Sequence Analyzer toolbar or select **Debug»Sequence Analyzer»Analyze <project filename>**. You can also select **Project»Analyze** in the sequence analyzer application.



Note Use the status bar at the bottom of the Analysis Results pane in the sequence editor and at the bottom of the Messages tab in the sequence analyzer application to view the progress of the analysis.



Note At any time during analysis, you can click the **Stop Analysis** button on the Sequence Analyzer toolbar or select **Debug»Sequence**

Analyzer»Stop Analysis in the sequence editor or select **Project»Stop Analysis** in the sequence analyzer application. The sequence analyzer finishes analyzing the current object before stopping the analysis and shows the current list of messages on the Analysis Results pane in the sequence editor and on the Messages tab in the sequence analyzer application.

6. (Optional) When you complete an analysis, you can save an XML report file to use in an external viewer for comparison. In the sequence editor, click **Generate Analysis Results** on the Analysis Results pane or select **Project»Generate Analysis Report** in the sequence analyzer to save the messages for the most recent analysis in an XML report file to use in an external viewer. The XML file uses the AnalyzerReportViewer.xsl style sheet, located in the `<TestStand>\Components\Stylesheets\Analyzer` directory. When you generate a report file, the sequence analyzer provides a snapshot of the most recent analysis, including a list of the analyzed files, a list of the rules and settings used during the analysis, and a list of all the generated messages.



Note Use the XML Packaging Utility to package the files necessary to view the XML report on a computer that does not have TestStand installed.

7. Use the Show and Group By drop-down lists to filter the generated messages on the Analysis Results pane in the sequence editor and on the Messages tab in the sequence analyzer application. Right-click each message and select an option from the Analyzer Messages tab context menu in the sequence editor or the Messages tab context menu in the sequence analyzer application to resolve the message. For each active message, you can perform the following actions:
 - a. Go to the location of the object in the TestStand file that caused the message and resolve the issue.
 - b. Ignore the issue for subsequent analysis and remove it from the Active Messages view. Select this option only when you are certain the issue will not cause a run-time error.
 - c. Mark the message as fixed and remove it from the Active Messages view.
 - d. Disable or configure the corresponding rule on the Rules pane of the sequence analyzer.



Note You can save the project file at any time when you are resolving messages.

You can repeat these steps to cycle through the analysis, report generation, and message resolution stages until you attain acceptable results.

Related concepts:

- [Search Directories](#)
- [TestStand Sequence Analyzer Rules Descriptions](#)
- [TestStand Sequence Analyzer Application](#)

Related tasks:

- [Creating Custom Rules in the TestStand Sequence Analyzer](#)

Related information:

- [Analysis Results Pane](#)
- [Current Sequence Analyzer Project Window](#)

TestStand Sequence Analyzer Rules Descriptions

The following sections describe the built-in TestStand rules.

- General
- Statistics
- Analysis Errors
- Performance
- Best Practices

General

Rule	Severity	Description
Expressions must evaluate to a value of the expected	Error	Each expression property must not include syntax errors and must evaluate to a value of the type the property expects. Syntax errors and incorrect value types can cause errors at run time. If you are certain that the expression will evaluate correctly at run time, such as

Rule	Severity	Description
type		when you dynamically create variables or when you propagate variables, use the #NoValidation directive in the expression to suppress the error.
Step properties must have valid values	Error	All step properties must have valid values. Invalid property values can cause errors at run time. Use the Step Settings pane to set the property to a valid value. Ignore this message if the property will be set dynamically to a correct value at run time.
Files must be found on disk	Error	File paths must refer to existing files on disk. TestStand can report errors at run time if it cannot find the file the path specifies. Correct the problem by correcting the path or by creating the file on disk. You can safely ignore this message if the file is created dynamically at run time.
Files should be found on disk	Warning	File paths should refer to existing files on disk. TestStand does not report errors at run time for these files, but the path specified cannot be found. Correct the problem by changing the path to point to the correct file or by creating the file on disk. Ignore this message if the file is created dynamically at run time.
Code modules must be able to load	Error	TestStand must be able to load all code modules without error. TestStand reports errors at run time when it cannot load a code module. Correct this problem by editing the step in the sequence editor.
Code modules should load without warnings	Warning	TestStand should be able to load all code modules without warnings. Module load warnings are issues that occur when trying to load a code module that might or might not be fatal or cause unexpected behavior. Correct this problem by editing the step in the sequence editor.
Code modules must have expected prototype	Error	The prototype for the code module must match the prototype the step module specifies. TestStand reports errors at run time when it loads a code module with an incompatible prototype. Correct the problem by editing the step module in TestStand to match the code module or by changing the code module prototype to match the step module.
Remote computer must be on the network	Error	Step properties that refer to remote computers must specify a valid network computer by IP address or by computer name. Ignore this message if the computer will exist at run time.
Steps must	Error	Steps with block structures, such as the Flow Control steps and the

Rule	Severity	Description
have expected block structure		Auto Schedule step, must have matching End steps, and End steps must have a matching Begin step. Other block requirements exist for Select/Case and Auto Schedule/Use Auto Scheduled Resource steps. TestStand reports errors at run time for invalid block structures.
Variables should be used	Warning	Unused variables and parameters add unnecessary complexity and extraneous information to sequence files. Delete variables if you never access them. You can ignore this rule for variables that you access in code modules or that you access using a lookup string with the TestStand API.
Sequences should be used	Warning	Unused sequences add extraneous information to sequence files and needlessly increase the size of the file. Delete sequences if you never use them. You can ignore this error for sequences that you call by expression or if you call the sequence from a sequence file that was not analyzed.
Steps should have Normal run mode	Warning	The run mode option for step should be set to Normal instead of Force Pass, Force Fail, or Skip to ensure that tests are performed correctly. Correct the setting on the Run Options Panel of the Step Settings pane in the TestStand Sequence Editor. Ignore this message if a step is intentionally set to this run mode.
Sequence calls should match the sequence prototype	Warning	The prototype for the sequence that a sequence call step calls should match the prototype the step specifies. You can ignore this rule for sequence calls in which you intend to pass a different set of parameters and if you check the parameters in the called sequence.
Find file paths that expressions specify	Information	Files that path expressions reference might not be included in the set of files to analyze. This rule provides a list of those expressions. You might need to manually add files expressions reference to the workspace for correct deployment.
All escape sequences used in expressions must be valid	Error	The sequence analyzer checks expressions for valid escape sequences. Improperly escaping paths usually leads to using invalid escape sequences, which are rarely intended.
Expression error caused by missing dynamic property	Warning	Expressions that use dynamic properties, such as RunState, will evaluate with errors at edit time. Ensure that the variable referenced will exist at run time. This message occurs when an expression evaluation error occurs because of a missing variable that starts with RunState or Parameters and the parameter is a container or object

Rule	Severity	Description
		reference. If this variable will exist at run time, you can safely ignore this message.
Types should have the correct version	Warning	Ensure that all types have versions that will not conflict with the default versions TestStand installs. This message can occur when you have a copy of CommonResults that is less than 3.1.0.100. National Instruments recommends that you set the version of CommonResults to the default of 3.1.0.100 if it contains no subproperties and set the version of CommonResults greater than 3.1.0.100 if it contains subproperties. Failure to change the version might cause type conflicts when loading files.
Steps that call LabVIEW modules configured with LabVIEW 7.1.1 or earlier should reload module prototypes	Warning	The prototype information stored on steps that call LabVIEW code modules might not be compatible with LabVIEW 2012 or later. This warning occurs for a step that calls a LabVIEW code module and the step was last configured using LabVIEW 7.1.1 or earlier. The step calls the code module correctly, but TestStand might generate an error when executing the step using LabVIEW 2012 or later. National Instruments recommends that you reload the prototype for the step or select Tools»Update VI Calls to run the Update VI Calls tool on any sequence file that return this warning.
Delete subelements of an output array parameter if you intend to log the entire array	Warning	If you enable additional result logging for an output array parameter on a LabVIEW or .NET step and the parameter contains subelements, the step logs only the subelements and not the entire array. An array parameter contains subelements if you added them manually when you specified the code module or if the code module is a LabVIEW VI that defines default element values for the array. To log the entire array, delete all the parameter subelements when you specify the code module. Ignore this message if you intend to log only the parameter subelements.
Find expressions that suppress evaluation errors	Information	Expressions that use the #NoValidation directive to suppress semantic evaluation errors might contain actual run-time errors.
Disable result recording for all process model sequences except the	Warning	If you enable result recording for a process model sequence other than the MainSequence callback, On-The-Fly reporting might not work correctly. Disable result recording for all process model sequences except the MainSequence callback.

Rule	Severity	Description
MainSequence callback		
Property Loader source should be proper	Error	Property Loader source should exist. The source should be according to valid format.
Enumeration values must be valid	Error	Each instance of an enumeration point must have a value that the type defines as valid.
Code modules should be up-to-date	Warning	Code modules should be up-to-date. The Sequence Analyzer checks that the file modification time for the code module is more recent than those of source files specified in the adapter step.  Note The Sequence Analyzer only checks source code files, project files, and solution/workspace files specified in the adapter step. It does not perform general dependency checking on included header files, other source files used by the code module, etc.
Feature is not supported	Error	Feature is not supported in the current bitness of TestStand that you are using.

Statistics

Rule	Severity	Description
Count steps	Information	The total number of steps in all the analyzed files, including count per sequence, count per sequence file, and count per analysis.
Count sequences	Information	The total number of sequences in all the analyzed files, including count per sequence file and count per analysis.
Count sequence files	Information	The total number of sequence files analyzed.

Analysis Errors

Rule	Severity	Description
Internal Error	Error	The sequence analyzer uses this rule to report internal errors that occur during analysis. These messages usually indicate a serious problem with a component of the sequence analyzer. Report the information to the component owner or to National Instruments. You cannot disable this rule.
File Open Error	Error	The sequence analyzer uses this rule to report errors that occur when opening files during analysis. You cannot disable this rule.
File Locked Error	Warning	The sequence analyzer uses this rule to report errors for files that it could not analyze because the files are locked and not viewable. You cannot disable this rule.

Performance

Rule	Severity	Description
Watch expressions are enabled	Warning	Using watch expressions during execution can negatively affect performance and cause slower execution speed. Use watch expressions only when debugging. Select Debug»Breakpoints/Watch Expressions to remove unused watch expressions.

Best Practices

Rule	Severity	Description
Avoid using absolute paths	Warning	Absolute paths make moving files among test systems difficult. Use relative paths to specify files.
All file paths must have a valid format	Error	All file path variables must contain valid syntax. The sequence analyzer verifies only that the syntax of the path is correct and does not confirm that a file exists at each location.

Related concepts:

- [Expressions](#)
- [Code Modules](#)

Viewing Examples in the Sequence Analyzer

Complete the following steps to show the example rules and analysis modules that use LabVIEW, LabWindows/CVI, Microsoft Visual C#, and Microsoft Visual C++ modules in the TestStand Sequence Analyzer.

1. Launch the Configure Sequence Analyzer Available Rules dialog box.
2. Click the **Import** button to launch the Import Items from File dialog box.
3. Click the **Browse** button and browse to `<TestStand Public>\Examples\TestStand Debugging Features\Custom Analyzer Rules - <RuleName>\<Environment>` directory and select the example rules file (`.tsarules`) you want to import in the **Select File to Import From** control, where `<RuleName>` is the rule name and `<Environment>` is one of the following:
LabVIEW, CVI, CSharp, or VC.
4. Enable all example rules and analysis modules in the **Select Items to Import** list.
5. Click the **Import** button to add the example rules and analysis modules to all new and existing sequence analyzer projects on the computer.
6. Click **Close** in the Import Items from File dialog box to return to the Configure Sequence Analyzer Available Rules dialog box.

The sequence analyzer adds the example rules and analysis modules to the **CustomRules.tsarules** section on the Rules tab and on the Analysis Modules tab of the Configure Sequence Analyzer Available Rules dialog box and adds an Example category to the Rules pane of the Current Sequence Analyzer Project window or the Rules tab of the stand-alone sequence analyzer application.

You can examine the source code for the example rules by viewing the files in the `<TestStand Public>\Examples\AnalyzerCustomRules` directory.



Note In addition to the sequence analyzer examples, National Instruments provides the source code for the built-in analysis rules in the `<TestStand>\Components\Analyzer\NIAnalysisModules` directory. The source code for the built-in analysis rules is appropriate for use as an example of how to use the sequence analyzer API, but is not recommended as a starting point for developing new rules.

Related concepts:

- [Search Directories](#)

Related tasks:

- [Creating Custom Rules in the TestStand Sequence Analyzer](#)

Related information:

- [TestStand Sequence Analyzer Environment Reference](#)
- [Configure Sequence Analyzer Available Rules Dialog Box](#)
- [Current Sequence Analyzer Project Window](#)
- [Import Items from File Dialog Box](#)

Creating Custom Rules in the TestStand Sequence Analyzer

You can customize the TestStand Sequence Analyzer by creating custom rules and analysis modules.

A rule defines a single task to perform during analysis, such as ensuring that the prototype of each code module is up to date or ensuring that each expression specified evaluates to the correct type and does not include any syntax errors. An analysis module is a VI, a class method in a .NET assembly, or a function in a DLL the sequence analyzer invokes to implement analyzer rules. Analysis modules perform the checks the rules specify and generate messages which the sequence analyzer stores in the analyzer project.

Analysis module specifications specify the analysis modules to call. Analysis module specifications also specify when the sequence analyzer calls the analysis module, the types of objects to which the analysis modules apply, and whether to call the analysis module when transitioning from one type of object to another. The sequence analyzer stores custom rules and analysis module specifications in `<TestStand Application Data>\Cfg\CustomRules.tsarules`. Refer to the Analyzer Examples for more information about custom rules and analysis modules.

Complete the following steps to create a custom analyzer rule and a corresponding analysis module.

1. If you are using the sequence analyzer in the TestStand Sequence Editor, click the

Sequence Analyzer Options button on the Sequence Analyzer toolbar or select **Debug»Sequence Analyzer»Sequence Analyzer Options** to launch the Sequence Analyzer Options dialog box.



Note The sequence editor requires an enabled ConfigApp user privilege to launch the Configure Available Rules dialog box. The built-in TestStand Developer and Administrator user groups grant this privilege by default.

2. If you are using the stand-alone sequence analyzer application, click the **Available Rules** button in the Sequence Analyzer Options dialog box or select **Edit»Available Rules** to launch the Configure Sequence Analyzer Available Rules dialog box.
3. Complete the following steps to create or edit a custom rule in the Configure Sequence Analyzer Available Rules dialog box.



Note The set of rules available on a computer is the same for all analyzer projects you open on that computer. When you use the Configure Sequence Analyzer Available Rules dialog box to add a new rule to the computer and then open an older project, the sequence analyzer automatically adds the new rule to the project. Similarly, the sequence analyzer removes from projects any rules that you removed using the Configure Sequence Analyzer Available Rules dialog box.

- a. Select the Rules tab and expand the **CustomRules.tsarules** section.
- b. Click **New Rule** button to the right of the <Insert New Rule> Rule ID to launch the Edit Rule dialog box.
- c. On the General tab of the Edit Rule dialog box, enter or select values for the Rule ID, Category, Severity, Name, and Description controls. You can also use the Rules tab of the Configure Sequence Analyzer Available Rules dialog box to edit the general settings of custom rules.



Note Select a Rule ID that is unique to the rules on your computer and the rules on any other computers to which you might deploy the rule. National Instruments recommends that you add your company name as a prefix to the Rule ID to prevent potential conflicts with other custom Rule IDs.

- d. (Optional) On the Advanced tab of the Edit Rule dialog box, enter or select values for the Resource String Category and Configuration Option controls to

specify options for localization and configuration. You can add custom rule settings by using the Edit Rule dialog box to launch a standard sequence analyzer dialog box in which users configure custom rules or by creating a custom rule configuration module to launch a dialog box you create in which users configure custom rules.

- e. Click **OK** to save the changes and close the Edit Rule dialog box.
4. Use LabVIEW, LabWindows/CVI, Microsoft Visual C#, or Microsoft Visual C++ to create an analysis module that implements one or more rules.
5. Complete the following steps to add a custom analysis module to the **CustomRules.tsarules** section on the Analysis Modules tab of the Configure Sequence Analyzer Available Rules dialog box.
 - a. Select the Analysis Modules tab and expand the **CustomRules.tsarules** section.
 - b. Click the **New Analysis Module** button to the right of the <Insert New Analysis Module> analysis module to launch the Edit Analysis Module dialog box.
 - c. In the Edit Analysis Module dialog box, enter the path to the module, a VI, .NET assembly, or DLL and select the kinds of objects, object types, and transition settings for the custom analysis module.
 - d. If you specify a VI as the analysis module and you want to debug the VI, disable the **Run VI in LabVIEW Run-Time Engine** option. You must also enable the **Development System** option in the Select or Type Which LabVIEW Server to Use section of the LabVIEW Adapter Configuration dialog box.
 - e. Click **OK** to save the changes and close the Edit Analysis Module dialog box.

You can deploy custom rules and analysis modules for use in analysis projects or to analyze individual files on other computers on which TestStand is installed.

Related concepts:

- [Code Modules](#)
- [Search Directories](#)
- [Adding Users and Setting Privileges](#)
- [Creating Analysis Modules for Custom Rules](#)
- [Creating Rule Configuration Modules in the TestStand Sequence Analyzer](#)
- [Debugging Analysis Modules and Rule Configuration Modules](#)
- [TestStand Sequence Analyzer Application](#)

Related tasks:

- [Adding Rule Settings for Custom Rules in the TestStand Sequence Analyzer](#)

- [Deploying Custom Rules and Analysis Modules](#)

Related information:

- [Configure Sequence Analyzer Available Rules Dialog Box](#)
- [Edit Analysis Module Dialog Box](#)
- [Edit Rule Dialog Box](#)
- [LabVIEW Adapter Configuration Dialog Box](#)
- [Sequence Analyzer Options Dialog Box](#)

Editing Custom Rules and Analysis Modules

When you edit a custom rule and corresponding analysis module, the changes apply to all projects.

Before you edit a VI, .NET assembly, or DLL analysis module, you must unload the analysis module from memory. You can unload all analysis modules by right-clicking anywhere on the Analysis Modules tab of the Configure Sequence Analyzer Available Rules dialog box and selecting **Unload All Modules** from the context menu. In the TestStand Sequence Editor, you can also select **File»Unload All Modules** to unload all modules, including analysis modules, from memory.

Related information:

- [Configure Sequence Analyzer Available Rules Dialog Box](#)

Localizing Custom Rules in the TestStand Sequence Analyzer

Complete the following steps to localize custom rules you create.

1. Create a string resource file (.ini file) in the `<TestStand Public>\Components\Language` directory.
2. To localize a custom rule category, add an `ANALYZER_RULE_CATEGORIES` section to the file and create tags and values for the custom rule category.
3. To localize rule strings, add a new section to the file and create tags and values for the custom rule name, description, and enumeration item names.
4. In the Resource String Category control on the Advanced tab of the Edit Rule dialog box, enter the name of the custom rule category section you created for the tags of

the custom rule name, description, and enumeration item names.

Related concepts:

- [Creating String Resource Files](#)
- [Search Directories](#)

Related information:

- [Edit Rule Dialog Box](#)

Creating Analysis Modules for Custom Rules

NI recommends that you copy an analysis module template from the `<TestStand>\Components\Analyzer\Templates\<Environment>` directory to the `<TestStand Public>\Components\Analyzer\<YourRuleName>` directory, where `<Environment>` is LabVIEW, CVI, CSharp, or VC, or LabVIEW NXG and `<YourRuleName>` is the descriptive name or the unique ID of the custom rule. NI recommends also using your company name as part of the directory name to avoid potential conflicts with other custom analysis modules. After you copy the template files, search for the text `Template` in each source file and follow the instructions in the file.

Related concepts:

- [Search Directories](#)
- [Accessing Rule Configuration Data in Rule Configuration Modules and Analysis Modules](#)
- [Accessing Rule Settings in Analysis Modules](#)
- [Debugging Analysis Modules and Rule Configuration Modules](#)

Related tasks:

- [Creating Custom Rules in the TestStand Sequence Analyzer](#)

Prototypes for Analysis Modules

The TestStand Sequence Analyzer requires that analysis modules use the following

specific prototypes.

LabVIEW

The VI connector pane must use an **ActiveX Refnum** input with the ActiveX class set to `TSAnalyzer.AnalysisContext` and can optionally include an **Error Out** cluster output or a Boolean, an I32 numeric, and a string output, which represent the unbundled elements of an **Error Out** cluster. You can use the Unbundle by Name function in LabVIEW to obtain the individual elements of the **Error Out** cluster.

LabWindows/CVI

The function prototype must be one of the following:

```
void Function(CAObjHandle analysisContext);
```

```
bool Function(CAObjHandle analysisContext, char
errorMessage[]);
```

```
int Function(CAObjHandle analysisContext, char
errorMessage[]);
```

where `analysisContext` is an input that contains an instance of the `TSAnalyzer.AnalysisContext` ActiveX class, and `errorMessage` is a string buffer of 1024 characters in which the function returns error messages for the analysis module code.

Microsoft Visual C#

The method prototype must be one of the following:

```
void
Method(NationalInstruments.TestStand.Interop.SequenceAnalyzer.Analy
analysisContext);
```

```
bool
Method(NationalInstruments.TestStand.Interop.SequenceAnalyzer.Analy
analysisContext, out string errorMessage);
```

where `analysisContext` is an input that contains an instance of the `AnalysisContext` COM interop class and `errorMessage` is an output string in which the method returns error messages for the analysis module code.

Microsoft Visual C++

The function prototype must be one of the following:

```
void Function(TSAnalyzer::AnalysisContext
*analysisContext);
```

```
bool Function(TSAnalyzer::AnalysisContext *analysisContext,
char errorMessage[]);
```

```
int Function(TSAnalyzer::AnalysisContext *analysisContext,
char errorMessage[]);
```

where `analysisContext` is an input that contains an instance of the `TSAnalyzer.AnalysisContext` ActiveX class, and `errorMessage` is a string buffer of 1024 characters in which the function returns error messages for the analysis module code.

Related concepts:

- [Implementing Analysis Modules](#)

Related information:

- [AnalysisContext](#)

Implementing Analysis Modules

The sequence analyzer passes an `AnalysisContext` object to the analysis module. After calling a VI, the sequence analyzer checks the **Error Out** cluster output and reports an Internal Error analysis message if the status indicates that an error occurred. After calling a .NET method, the sequence reports an Internal Error analysis message if the method returns `False`. After calling a DLL function, the sequence analyzer checks the return value and reports an Internal Error analysis message if the **bool** value is `False`.

or if the **int** value is not 0. The Analysis Results pane in the TestStand Sequence Editor and the Messages tab in the stand-alone sequence analyzer application include the error message the analysis module returns for the most recent analysis of the current project.

Analysis modules use the `AnalysisContext` properties and methods to analyze individual objects. The sequence analyzer does not maintain or recognize any relationships between analysis modules and rules. The sequence analyzer calls all analysis modules regardless of which rules are disabled, but the sequence analyzer discards messages reported for disabled rules. To improve performance when a rule is disabled, ensure that an analysis module first checks if the rule is enabled before performing any analysis. Pass the rule ID to the `AnalysisContext.GetRuleConfiguration` method to get the `RuleConfiguration` for the rule and check the `RuleConfiguration.Enabled` property. Pass the rule ID to the `AnalysisContext.NewMessage` method to create a new message for the rule, and call the `AnalysisContext.ReportMessage` method to report the message to the sequence analyzer. Use the `AnalysisContext.File` and `AnalysisContext.Object` properties to get the analyzed file and object.

The sequence analyzer calls an analysis module only for the kinds of objects you specify in the Edit Analysis Module dialog box. Each time the sequence analyzer makes a transition between analyzing one type of object to analyzing a different type of object, it calls analysis modules configured to be called for that type of transition. If you specify analysis transitions in the Edit Analysis Module dialog box, the sequence analyzer calls the analysis module at those transitions. Use the `AnalysisContext.Transition` property to determine which transition is in effect.

Use the `AnalysisContext.GetRuleAnalysisData` method to get an empty `PropertyObject` container to which you add subproperties to store data associated with a rule during the current analysis session. You must use the `GetRuleAnalysisDataOption_Lock` option if the analysis module modifies the data.

If you use global variables during analysis, use the following techniques to help ensure data integrity:

- Use locks to protect access to the global data by preventing the sequence analyzer from calling analysis modules from multiple threads at the same time.
- Use the `AnalysisContext.AnalysisId` property to get a unique identifier for the

current analysis session that you can use to index into a global table to avoid interference between simultaneous analysis sessions.

- Use the `AnalysisTransition_BeforeSystem` and `AnalysisTransition_AfterSystem` transitions to initialize and clean up global data.

When you stop the analysis before it completes, the sequence analyzer waits for the currently executing analysis module to complete. If the sequence analyzer has called any analysis modules for any **Before** analysis transitions, the sequence analyzer calls the same analysis modules for the corresponding **After** analysis transitions. Analysis modules generally should not report messages after you stop analysis. If the analysis module reports messages during the **After** analysis transitions, check the `AnalysisContext.StopRequested` property to determine whether users stopped the analysis so you can avoid reporting the messages.

Use the `AnalysisContext.Utilities` property to access the `AnalysisUtilities` methods, which perform common types of checks, such as validating expressions, paths, and remote hosts, or enabling automatic property checking. The built-in analysis modules perform some of these checks on all instances of certain types, such as validating expressions and paths. To disable the built-in checking for subproperties of certain objects for a custom rule, set the `AnalysisUtilitiesAutomaticPropertyCheckingEnabled` property to `False` in the custom analysis module the sequence analyzer calls for those objects. For instance, to disable automatic checking of subproperties of a step type that you implemented, set the `AnalysisUtilitiesAutomaticPropertyCheckingEnabled` property to `False` in the custom analysis module the sequence analyzer calls for instances of the custom step type.

Related concepts:

- [Expressions](#)
- [Prototypes for Analysis Modules](#)

Related information:

- [RuleConfiguration](#)
- [AnalysisContext](#)
- [PropertyObject](#)
- [GetRuleAnalysisDataOptions](#)

- [AnalysisUtilities](#)
- [AnalysisTransition](#)

Making a Custom Rule Configurable in the TestStand Sequence Analyzer

If you want users to configure the custom rule you create, you must select one of the following rule configuration options on the Advanced tab of the Edit Rule dialog box:

- **Use Rule Settings**—Define a set of Boolean, number, string, or enumeration settings that users can set in the Rule Settings dialog box, which the TestStand Sequence Analyzer launches when a user clicks the **Settings** button on the Rules pane or tab. The standard dialog box stores the settings for the rule that the analysis module accesses. In general, this option is easier to use because you do not have to write a rule configuration module or dialog box.
- **Use Configuration Module**—Create a rule configuration module that launches a dialog box the sequence analyzer launches when a user clicks the **Settings** button on the Rules pane or Rules tab. The dialog box you create stores custom data for the rule that the analysis module accesses. Select this option in the following situations:
 - You prefer a custom look for the dialog box users use to configure the rule.
 - You need to store data more complex than Boolean, number, string, or enumeration values, such as a list of string values.
 - You need to use custom controls, tree controls, or list controls in the dialog box.

Related concepts:

- [Accessing Rule Settings in Analysis Modules](#)
- [Creating Rule Configuration Modules in the TestStand Sequence Analyzer](#)

Related information:

- [Edit Rule Dialog Box](#)
- [Rule Settings Dialog Box](#)

Adding Rule Settings for Custom Rules in the TestStand Sequence Analyzer

Complete the following steps to add rule settings for users to configure custom rules.

1. On the Advanced tab of the Edit Rule dialog box, select **Use Rule Settings** from the **Configuration Option** ring control.
2. Click the **Add** button to add a new setting.
3. Use the Rule Settings table to edit the name, type, and default value for the new setting. Rule settings can have number, string, Boolean, or enumeration types. Click the **Enums** button to launch the Edit Rule Settings Enumerations dialog box, in which you create new rule settings enumeration types to use for rule settings.



Note When you add rule settings to a custom rule, the Rules pane or tab displays a **Settings** button next to the rule in the Edit Rule dialog box. Click the **Settings** button to launch the Rule Settings dialog box.

- Advanced tab

Accessing Rule Settings in Analysis Modules

Edit Rule Settings Enumerations dialog box

Edit Rule dialog box

Rules pane

Rule Settings dialog box

Rules tab

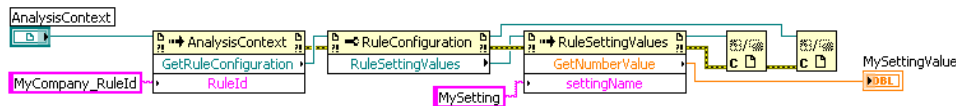
Accessing Rule Settings in Analysis Modules

When you select Use Rule Settings from the Configuration Option ring control on the Advanced tab of the Edit Rules dialog box to use rule settings during analysis, the analysis module uses the `AnalysisContext.GetRuleConfiguration` method to retrieve the rule configuration and then uses the `RuleConfiguration.RuleSettingValues`

property to access the setting values. Use the `RuleSettingValues.GetBooleanValue`, `RuleSettingValues.GetStringValue` and `RuleSettingValues.GetNumberValue` methods to obtain individual setting values.

The following examples show how to access a number rule setting value in LabVIEW, LabWindows/CVI, Microsoft Visual C#, and Microsoft Visual C++.

LabVIEW



LabWindows/CVI

```
TSAnalyzerObj_RuleConfiguration ruleConfiguration = 0;
TSAnalyzerObj_RuleSettingValues ruleSettingValues = 0;
double mySettingValue = 0.0;
tsErrChk(TSAnalyzer_AnalysisContextGetRuleConfiguration
(analysisContext, &errorInfo, "MyCompany_RuleId",
&ruleConfiguration));
tsErrChk(TSAnalyzer_RuleConfigurationGetRuleSettingValues(ruleConf
&errorInfo, &ruleSettingValues));
tsErrChk(TSAnalyzer_RuleSettingValuesGetNumberValue(ruleSettingValu
&errorInfo, "MySetting", &mySettingValue));
```

Error:

```
CLEAR(ruleConfiguration, CA_DiscardObjHandle);
CLEAR(ruleSettingValues, CA_DiscardObjHandle);
```

Microsoft Visual C#

```
RuleConfiguration ruleConfiguration =
analysisContext.GetRuleConfiguration("MyCompany_RuleId");
double mySettingValue =
ruleConfiguration.RuleSettingValues.GetNumberValue("MySetting");
```

Microsoft Visual C++

```
TSAnalyzer::RuleConfigurationPtr ruleConfiguration =
analysisContext->GetRuleConfiguration("MyCompany_RuleId");
double mySettingValue = ruleConfiguration-
>RuleSettingValues->GetNumberValue("MySetting");
```

Related information:

- [AnalysisContext](#)
- [RuleConfiguration](#)
- [RuleSettingValues](#)
- [Edit Rule Dialog Box](#)

Creating Rule Configuration Modules in the TestStand Sequence Analyzer

Select **Use Configuration Module** from the Configuration Option ring control on the Advanced tab of the Edit Rule dialog box to use a rule configuration module. The TestStand Sequence Analyzer calls the configuration module when the user clicks the Settings button next to the rule on the Rules pane or tab.

National Instruments recommends that you copy a rule configuration module template from the <TestStand>\Components\Analyzer\Templates\<Environment> directory to the <TestStand Public>\Components\Analyzer\<YourRuleName> directory, where Environment is LabVIEW, CVI, CSharp, or VC and YourRuleName is the descriptive name or the unique ID of the custom rule. National Instruments recommends also using your company name as part of the directory name to avoid potential conflicts with other custom analysis modules.

(LabVIEW) After you copy the template files, rename the VIs using the rule ID to avoid potential conflicts with other analysis module VIs, and enable the VIs to perform rule checking.

(LabWindows/CVI, Microsoft Visual C#, Microsoft Visual C++) After you copy the template file, search for the text `Template` in each source file and follow the

instructions in the file.

The TestStand Sequence Analyzer requires that rule configuration modules use the following specific prototypes.

LabVIEW

The VI connector pane must use an **ActiveX Refnum** input with the ActiveX class set to `TSAnalyzer.RuleConfigurationContext` and can optionally include an **Error Out** cluster output or a Boolean, an I32 numeric, and a string output, which represent the unbundled elements of an **Error Out** cluster. You can use the Unbundle by Name function in LabVIEW to obtain the individual elements of the **Error Out** cluster.

LabWindows/CVI

The function prototype must be one of the following:

```
void Function(CAObjHandle ruleConfigurationContext); bool
Function(CAObjHandle ruleConfigurationContext, char
errorMessage[]); int Function(CAObjHandle
ruleConfigurationContext, char errorMessage[]);
```

where `ruleConfigurationContext` is an input that contains an instance of the `TSAnalyzer.RuleConfigurationContext` ActiveX class, and `errorMessage` is a string buffer of 1024 characters in which the function returns error messages for the rule configuration module code.

Microsoft Visual C#

The method prototype must be one of the following:

```
void
Method(NationalInstruments.TestStand.Interop.SequenceAnalyzer.RuleC
ruleConfigurationContext); bool
Method(NationalInstruments.TestStand.Interop.SequenceAnalyzer.RuleC
ruleConfigurationContext, out string errorMessage);
```

where `ruleConfigurationContext` is an input that contains an instance of the

RuleConfigurationContext COM interop class and errorMessage is an output string in which the method returns error messages for the rule configuration module code.

Microsoft Visual C++

The function prototype must be one of the following:

```
void Function(TSAnalyzer::RuleConfigurationContext
*ruleConfigurationContext); bool
Function(TSAnalyzer::RuleConfigurationContext
*ruleConfigurationContext, char errorMessage[]); int
Function(TSAnalyzer::RuleConfigurationContext
*ruleConfigurationContext, char errorMessage[]);
```

where ruleConfigurationContext is an input that contains an instance of the TSAnalyzer.RuleConfigurationContext ActiveX class, and errorMessage is a string buffer of 1024 characters in which the function returns error messages for the rule configuration module code.

Related concepts:

- [Search Directories](#)
- [Accessing Rule Configuration Data in Rule Configuration Modules and Analysis Modules](#)

Related information:

- [Edit Rule Dialog Box](#)
- [RuleConfigurationContext](#)

Accessing Rule Configuration Data in Rule Configuration Modules and Analysis Modules

The TestStand Sequence Analyzer passes a RuleConfigurationContext object to the rule configuration module. After calling a VI, the sequence analyzer checks the **Error Out** cluster output and reports an error if the status indicates that an error occurred.

After calling a function, the sequence analyzer checks the return value and reports an error if the **bool** value is `False` or if the **int** value is not 0.

To save custom rule settings, the rule configuration module first uses the `RuleConfigurationContext.RuleConfiguration` property to retrieve the rule configuration. The rule configuration module adds setting values to the `RuleConfiguration.ConfigurationData` `PropertyObject`, which is initially an empty container. When the user configures the rule, the rule configuration module adds subproperties to the `PropertyObject` that corresponds to the custom rule settings.

To use custom rule settings during analysis, the analysis module first uses the `AnalysisContext.GetRuleConfiguration` method to retrieve the rule configuration and then accesses the `RuleConfiguration.ConfigurationData` `PropertyObject`, which is an empty container if the user did not configure the rule. The analysis module uses the default setting values in this case. Otherwise, the analysis module retrieves the setting values from the subproperties of the `PropertyObject`. Set the `RuleConfiguration.Description` property in a rule configuration module to include a description of the setting values. This enables users to view the rule settings through the rule descriptions in the Rules pane or tab and populates generated reports with a description of the setting values.

Related information:

- [RuleConfigurationContext](#)
- [RuleConfiguration](#)
- [AnalysisContext](#)
- [PropertyObject](#)

Debugging Analysis Modules and Rule Configuration Modules

You can debug your analysis modules and rule configuration modules by setting a breakpoint in the module and then calling the module from the TestStand Sequence Analyzer. Complete the following steps to debug an analysis module or rule configuration module, depending on the application you used to create the module.

Related information:

- [Edit Analysis Module Dialog Box](#)

- [Edit Rule Dialog Box](#)

Debugging Analysis Modules and Rule Configuration Modules in LabVIEW

1. Disable the **Run VI in LabVIEW Run-Time Engine** option on the Edit Analysis Module dialog box for an analysis module or on the Advanced tab of the Edit Rule dialog box for a rule configuration module.
2. Configure the LabVIEW Adapter to use the LabVIEW development system.
3. In LabVIEW, open the VI and set breakpoints in the block diagram for debugging. Refer to the LabVIEW Help for more information about using breakpoints.

Debugging Analysis Modules and Rule Configuration Modules in LabWindows/CVI

1. Build a debuggable DLL in LabWindows/CVI.
2. Set a breakpoint in the analysis module or rule configuration module function.
3. In LabWindows/CVI, select **Run»Select External Process** in the Project window to identify the executable for the sequence editor or the standalone sequence analyzer application and then select **Run»Debug <executable name>** to start debugging the executable.

Debugging Analysis Modules and Rule Configuration Modules in Microsoft Visual C# and Microsoft Visual C++

1. Build a debug version of the DLL in Microsoft Visual Studio.
2. Set a breakpoint in the analysis module or rule configuration module function.
3. In Visual Studio, set the Command setting in the Debugging project settings to the executable for the sequence editor or the standalone sequence analyzer application and select **Debug»Start Debugging** to start debugging the executable.

Calling Rule Configuration Modules for Custom Rules

Complete the following steps to call a rule configuration module to launch a dialog box you create to configure a custom rule.

1. On the Advanced tab of the Edit Rule dialog box, select **Use Configuration Module** from the **Configuration Option** ring control.
2. Click the **Browse** button to navigate to the VI, .NET assembly, or DLL rule

configuration module you want to use.

3. If you selected a DLL, use the **Function** ring control to specify the function in the DLL to call.
4. If you selected a .NET assembly, use the **Class** and **Method** ring controls to specify the method in the .NET assembly to call.



Note When you add rule settings to a custom rule, the Rules pane or tab displays a Settings button next to the rule in the Edit Rule dialog box. Click the Settings button to launch the Rule Settings dialog box.

Related concepts:

- [Creating Rule Configuration Modules in the TestStand Sequence Analyzer](#)

Related information:

- [Edit Rule Dialog Box](#)
- [RuleConfigurationContext](#)
- [Rules Pane - Current Sequence Analyzer Project Window](#)

Deploying Custom Rules and Analysis Modules

You can deploy custom rules and analysis modules to use in analysis projects or to analyze individual files on other computers on which TestStand is installed. By default, when you import custom rules and analysis modules on a target computer, the TestStand Sequence Analyzer enables the rule for all new and existing projects on that computer. You must manually disable the rule in each project on the computer.

Complete the following steps to deploy custom rules and analysis modules by exporting the files from your computer, transferring the files to the target computer, and importing the files on the target computer.



Note If the rule ID of a custom rule you want to export conflicts with the rule ID of an existing rule on the target computer, the rule you export replaces the existing rule on the target computer. Similarly, if the path of a custom analysis module specification you want to export conflicts with an existing analysis module specification on the target computer, the custom analysis

module specification you export replaces the existing analysis module specification on the target computer. You can update custom rules and analysis modules you already imported on the target computer using this behavior.



Note If the search directories on the target computer differ from the search directories on the computer from which you are exporting files, the sequence analyzer might not find the custom analysis modules and rule configuration modules associated with the custom rules you export. You can change the search directories on the target computer, or you can copy the custom analysis modules and rule configuration modules to a different location on the target computer.



Note If you use LabVIEW to implement an analysis module and you disable the Use LabVIEW Run-Time Engine option on the Edit Analysis Module dialog box, the sequence analyzer uses the LabVIEW Adapter settings on the target computer to determine whether to run the VI in the LabVIEW development system or in the LabVIEW RTE. If you enable the Use LabVIEW Run-Time Engine option, the sequence analyzer uses the LabVIEW RTE on the target computer. The sequence analyzer reports an error if the necessary LabVIEW RTE is not installed on the target computer.

1. On your computer, launch the Configure Sequence Analyzer Available Rules dialog box.
2. Click the **Export** button to launch the Export Items to File dialog box.
3. In the Export Items to File dialog box, enable the custom rules and analysis module specifications you want to export in the **Select Items to Export** list.
4. Click the **Export** button to launch a file dialog box, in which you create or browse to the rules file (`.tsarules`) to use. By default the sequence analyzer saves the file in the `<TestStand Public>\Components\Analyzer` directory.
5. Click **Close** in the Export Items to File dialog box.
6. Copy the VIs, .NET assemblies, and DLLs for your custom analysis modules and rule configuration modules to the same location on the target computer. Also copy strings files and any other supporting files the modules require to the target computer.
7. Copy the rules file you created in step 4 on your computer to the same location on

the target computer.

8. On the target computer, launch the Configure Sequence Analyzer Available Rules dialog box.
9. Click the **Import** button to launch the Import Items from File dialog box.
10. Select the rules file you copied in step 7 in the **Select File to Import From** control.
11. Enable the custom rules and analysis modules you want to import in the **Select Items to Import** list.
12. Click the **Import** button to add the custom rules and analysis modules to the target computer.
13. Click **Close** in the Import Items from File dialog box to return to the Configure Sequence Analyzer Available Rules dialog box.

You can also use the Sequence Analyzer Install Utility application to automatically import custom rules and analysis modules without user interaction from a rules file you create and export to a target computer.



Note Even when you use the Sequence Analyzer Install Utility application, you must manually copy the VIs, .NET assemblies, DLLs, strings files, and any other supporting files the exported analysis modules and rule configuration modules require to the target computer.

Related concepts:

- [Search Directories](#)

Related information:

- [Configure Sequence Analyzer Available Rules Dialog Box](#)
- [Edit Analysis Module Dialog Box](#)
- [Export Items to File Dialog Box](#)
- [Import Items from File Dialog Box](#)

Sequence Analyzer Install Utility

Use the command-line Sequence Analyzer Install Utility application, `AnalyzerInstallUtil.exe`, located in the `<TestStand>\Bin` directory to automatically install or uninstall custom rules and analysis modules without user

interaction from a rules file you create and export to a target computer.

Use the following syntax to invoke the Sequence Analyzer Install Utility, which adds all new and replaces all existing custom rules and analysis modules from the rules file on the target computer and shows a list of the items installed:

```
AnalyzerInstallUtil.exe [-U] [-S] <MyRules.tsarules>
```

The [-U] option uninstalls from the target computer existing custom rules and analysis modules that correspond to items in the rules file. The tool removes from the target computer rules with the same rule ID and analysis modules with the same path and function as rules and analysis modules in the rules file.

The [-S] option instructs the tool to operate in silent mode and not list the items installed or uninstalled.

Related concepts:

- [Search Directories](#)

Converting LabVIEW Analysis Modules and Rule Configuration Modules to DLLs

Before you deploy LabVIEW analysis modules, consider building a DLL using a LabVIEW project. By deploying a LabVIEW-built DLL, you can avoid having to deploy multiple VIs that comprise the full hierarchy of the analysis modules and rule configuration modules. Refer to the LabVIEW Help for more information about building shared libraries, or DLLs. When you build a DLL with LabVIEW, you must specify the VI prototype the TestStand Sequence Analyzer expects.

Complete the following steps in the LabVIEW Define VI Prototype dialog box.

1. For the return value, change the **VI Output** to `code`, which is the **code** element of the **Error Out** cluster.
2. Delete the **code** parameter.
3. Delete the **status** parameter.

After you complete these steps, the **Function Prototype** control should include the following code:

```
int32_t VIName(IDispatch *AnalysisContext, char source[], int32_t len)
```

Execution Profiler

Select the  button to launch the Execution Profiler window to view and record duration of steps, code modules, and other resources a multithreaded TestStand system uses over a period of time.

You can review the recorded data in graphs and sortable tables to identify performance bottlenecks and design flaws and to gain insight into the behavior and timing of complex multithreaded systems. You can copy the information to external applications, such as Microsoft Word or Excel.

Related concepts:

- [Comparing Resource Usage Strategies](#)

Related information:

- [Synchronization Step Types](#)

Types of Information the Profiler Records

Types of Information the Profiler Records

Events

The Execution Profiler records events that contain information about the following:

- **Step and Step Type Module Execution Times**—The duration of code modules that steps call using any adapter, including subsequences called by Sequence Call steps.
- **Step and Step Type Module Load and Unload Times**—The time spent loading and unloading LabVIEW VIs, C/C++ dlls, .NET assemblies, and COM servers that steps and step types call.
- **Step Execution Times**—The complete execution duration for each step. This includes the code modules it calls, step looping, expression evaluation, result recording, post actions, switching, synchronization, and any other actions a step is

configured to perform.

- **Batch, UUT, and Lot Duration**—The time period spent testing each UUT, each Batch, and each Lot.



Note A Lot is a series of batches tested using the NI TestStand Semiconductor Module.

- **Synchronization Operations**—The time spent waiting and duration of use for all synchronization operations, including operations on Locks, Notifications, Queues, Semaphores, Batch Synchronization Sections, Auto Schedule Blocks, and explicit Waits.

For each event, the profiler records the time of the event, the TestStand thread and execution in which it occurs, the sequence file, and location in the file from which it initiates. The profiler also records the name of the item for the event. The item name depends on what triggers the event. For example, it could be the name of a TestStand lock for a Lock event, or a description of a code module call for a step module event. The profiler also records other information, such as the index of the test socket from which the event originates.

The Events table displays a list of all the events the Execution Profiler has recorded.

Operations

The profiler associates multiple related events for the same item in the same thread with the logical operation that the events record. For example, a Wait operation associates the two events that the profiler records when a thread waits: the Blocked event, which records when the thread begins waiting, and the Completed event, which records when the thread resumes execution.

The Operations table display these operations. You can also view operations in both of the graphs the profiler provides. Typically, you use the profiler primarily to view various operations and how they relate to each other in initiation time, duration, and by thread.

Items

Each operation the profiler records originates from a specific item. The item might be a

code module, a synchronization object, a UUT, a Batch, a Lot, or a Step. The Items table displays all the items for which the profiler has recorded information. You can view the number of times an item is used and the total time spent using an item. You can also select and navigate to the Operations or Events for selected items.

The Items table contains a list of all code modules, steps, UUTs, batches, lots, and synchronization objects for which the Execution Profiler has recorded events.

Threads

The profiler displays a list of every thread in which a profiler event occurs in the Threads table. You can use the Threads table to view the amount of time a thread has been in use by or blocked waiting for profiled operations. You can view the number of operations in and the number of items used by each thread. If a thread is the root thread for a test socket execution, the profiler displays its test socket index. If a profiler event occurs in a thread that is not a TestStand execution thread, the thread name displays Non-Execution Thread. Note that TestStand often uses non-execution threads to load and unload modules.

Executions

The profiler displays a list of the executions that contain the threads that the Threads table displays. You can select and navigate to the Threads in each execution. Threads that are not TestStand execution threads appear in the profiler as belonging to an item in the execution table named Not-An-Execution.

The Executions table contains a list of all executions that contain threads for which the Execution Profiler recorded events."

Related information:

- [Step Settings Pane](#)
- [Lock Step](#)

Item Names

Item Names

For a code module, the item name is a description of the module call. For UUTs, Batches, and Lots the item name includes the serial number or lot id, if specified. If it is needed to distinguish separate items, the item name includes an index value. For Steps, the item name is the name of the step. If multiple steps have the same name, the item name includes the unique identifier for the step.

If item names coincide for different types of operations, the profiler appends the type of operation to the item name, such as 'Lock' or 'Step'.

Locks and Other Synchronization Item Names

You typically name TestStand locks after the resources the locks protect, such as DMM1 or DC Power Supply. The profiler displays the name of the lock, or other synchronization object, as the profiler item name. Thus, if you follow the convention of naming your locks after the resources they protect, you can easily use the profiler to observe the usage of and contention for your physical and logical resources.

If you create a lock or another type of synchronization object with an empty name, TestStand names the object for you, such as Unnamed Synchronization Object N. TestStand also automatically selects an item name in the following cases:

- When you use the Use Lock to Allow Only One Thread at a Time to Execute the Step option on the Synchronization panel of the Properties tab of the Step Settings pane and not specifying an existing lock
- When you use the Batch Synchronization option on the Synchronization panel of the Properties tab of the Step Settings pane
- When you use a Wait step
- When you use a Use Auto Scheduled Resource step

In these cases, TestStand creates an item name by combining the unique ID of the step with the run-time ID of the file that contains the step. However, for readability, the profiler window shows the name of the step as the item name. If the step name is not a unique item name, the profiler appends the unique id for the step to the step name. For Use Auto Scheduled Resource steps that do not have unique names, the profiler

first attempts to make the step name unique by appending a description of the locks the step acquires.

Related information:

- [Step Settings Pane](#)
- [Lock Step](#)

Resource Names - Resource Usage Profiler

Resource Names

You typically name TestStand locks after the resources the locks protect, such as `DMM1` or `DC Power Supply`. When the Resource Usage Profiler displays information about a resource, such as the name of the resource, the profiler is actually describing the lock that protects the resource.

If you create a lock or another type of synchronization object with an empty name, TestStand names the object for you, such as `Unnamed Synchronization Object - <N>`. TestStand also automatically selects a resource name in the following cases:

- When you use the Use Lock to Allow Only One Thread at a Time to Execute the Step option on the Synchronization panel of the Properties tab of the Step Settings pane and not specifying an existing lock
- When you use the Batch Synchronization option on the Synchronization panel of the Properties tab of the Step Settings pane
- When you use a Wait step
- When you use a Use Auto Scheduled Resource step

In these cases, TestStand creates a resource name by combining the unique ID of the step with the run-time ID of the file that contains the step. However, for readability, the profiler window shows the name of the step as the resource name. If the step name is not a unique resource name, the profiler appends the unique resource name that TestStand creates to the step name. For Use Auto Scheduled Resource steps that do not have unique names, the profiler first attempts to make the step name unique by appending a description of the locks the step acquires.

Related information:

- [Step Settings Pane](#)
- [Use Auto Scheduled Resource Step](#)
- [Wait Step](#)

Determining Resource Usage

Locks and Resource Usage

The Execution Profiler cannot directly determine when a thread uses a logical or physical resource. Instead, the profiler records when a thread waits for, acquires, or releases a TestStand lock. Typical TestStand applications use TestStand locks to surround the use of logical and physical resources to guarantee that only one thread accesses the resources at a time. Because lock acquisition and resource usage correspond, viewing the time line of lock acquisitions provides a close approximation of actual resource usage. In addition, because threads wait to acquire locks rather than resources, viewing lock operations shows when threads block waiting for a resource to become available. If threads block excessively because of limited resources or a design flaw, viewing the associated lock operations can highlight the location of the performance issue.

A system might use resources you protect with mechanisms other than TestStand locks, including lower-level primitives, such as operating system critical sections and mutexes. To view the uses of resources with non-TestStand locks in the profiler, you must also surround the resources with TestStand locks.

Depending on the system, you might be able to place TestStand locks at a much higher level than the underlying thread synchronization. For example, if a step calls a code module you write that spends most of its time in a call to an instrument driver function that internally synchronizes thread access to the instrument, you can place a TestStand lock around the calling step to approximate the instrument driver synchronization in a way that is viewable in the profiler.

Use the Lock step type or the Use Lock to Allow Only One Thread at a Time to Execute the Step option to create and invoke TestStand locks. You can also call the TestStand API to create and invoke TestStand locks from within code modules you write.

Related concepts:

- [Code Modules](#)

Related information:

- [Lock Step](#)

Auto Scheduling and Thread Synchronization Operations

Auto Scheduling and Thread Synchronization Operations

When you use an Auto Schedule step, you also create sub-blocks of steps by using the Use Auto Scheduled Resource steps. Each Use Auto Scheduled Resource step specifies the resources, or locks, it must acquire to run the steps within the sub-block of the step. Before a thread can acquire the resources a Use Auto Scheduled Resource step specifies, the thread must first acquire exclusive access to the step itself.

The Execution Profiler shows an item that represents the Use Auto Scheduled Resource step so you can view the time that threads spend waiting to access the step. Select **View»Show Synchronization»Show Auto Scheduling** in the Execution Profiler window to remove the checkmark that appears to the left of the menu item to hide items that represent Use Auto Scheduled Resource steps.

You can also select **View»Show Synchronization»Show<Synchronization Type>** in the profiler window to selectively show or hide the following other types of TestStand thread synchronization operations:

- Locks
- Rendezvous
- Queue
- Notification
- Batch Synchronization
- Semaphore
- Wait Operations

The profiler shows thread synchronization operations for only the current operating system process. In a typical system, most or all TestStand locks and synchronization

operations are local to the current process. You can create cross-process synchronization operations by prefixing the resource name with an asterisk character. You can also create cross-machine operations by prefixing the resource name with a computer name. However, these non-local operations do not appear in the profiler.

Related information:

- [Auto Schedule Step](#)
- [Execution Profiler Window](#)
- [Use Auto Scheduled Resource Step](#)

Displaying Additional Timing Information

Displaying Additional Timing Information

If you want to view the timing of additional operations the system performs relative to the operations the Execution Profiler shows, surround the additional operations with TestStand locks, even if the resources involved do not need multiple thread access protection. For example, to display when a lengthy calculation takes place and how long it takes, you can surround the calculation with a TestStand lock named "My Lengthy Calculation".

If you add a lock for profiling purposes around an operation that does not need thread synchronization, you can use a unique lock in each thread to ensure that multiple threads can run the operation in parallel. When each thread uses its own lock, a thread can never block waiting for another thread to release the lock. You can create a lock unique to a thread by appending a name unique to the thread to the name of the lock you create. A convenient source of a name unique to a TestStand thread is the `Thread.UniqueThreadId` property, which is available in TestStand expressions and from the TestStand API. For example, you can surround a lengthy calculation with a lock named "My Lengthy Calculation - " + `RunState.Thread.UniqueThreadId`.

An alternative method to display information in the profiler is to call the `Engine.LogProfilerAction` method. For example, before starting an operation you want the profiler to display, you might execute an expression similar to:

```
RunState.Engine.LogProfilerAction("Calibration", "",
```

```
ThisContext, Str(RunState.Thread.Id),
RunState.Thread.DisplayName, "Measure Fixture Wiring
Delay", ProfilerState_InUse, "Delay Compensation", -1,
True, Nothing)
```

After the operation completes, you might execute an expression similar to:

```
RunState.Engine.LogProfilerAction("Calibration", "",
ThisContext, Str(RunState.Thread.Id),
RunState.Thread.DisplayName, "Measure Fixture Wiring
Delay", ProfilerState_Completed, "Delay Compensation", -1,
True, Nothing)
```

Programmatically Displaying and Configuring the Profiler

Programmatically Displaying and Configuring the Profiler

There is at most one profiler per TestStand application process. Accessing any static method or property of the class

`NationalInstruments.TestStandProfiler.Profiler` from the `<TestStand>\Tools\Profiler\$(Platform)\Profiler.dll` assembly, except for the `Close()` method and the `IsOpen` property, automatically creates the profiler if needed before performing the specified action. Call the `Close()` method to close the profiler window and discard its data. The `Comparing Resource Usage Strategies.seq` file in the `<TestStand>\Examples\Parallel Testing\Comparing Resource Usage Strategies` directory contains a `SequenceFileLoad` callback sequence that contains example steps that launch and configure the profiler.

The Profiler enables you to programmatically:

- Load and Save profiler data
- Clear the profiler window
- Specify what types of information the profiler collects and displays
- Start or stop data collection and data display
- Activate, select data, and copy information from a specified table or graph
- Set other options, such as time units, window size and position, and splitter bar location

For example, to isolate a section of your sequence for profiling, you might insert a step before the section that sets `Profiler.DataCollectionEnabled` and a step at the end of the section, which clears `Profiler.DataCollectionEnabled`.



Note To prevent the execution of steps that control the profiler from appearing within the profile data, you can mark them for exclusion from profiling. Refer to the [Excluding Items from Profiling](#) topic for more information.

Related concepts:

- [Search Directories](#)

Related information:

- [Execution Profiler Window](#)

Excluding Items from Profiling

Excluding Items from Profiling

You can prevent the profiler from recording events for Sequence Files, Sequences, Steps, and Executions. To do so, add a container property to the item's attributes named `NIProfiler`. Within that container, create a Boolean property named `Exclude`. Note that if you create these attributes programmatically, in the case of a `SequenceFile`, add the attributes to the `SequenceFile.Data` container rather than directly to the `SequenceFile` object.

Profiler Window Graphs

Profiler Window Graphs

The tabs in the top half of the Execution Profiler window display the following graphs:

- Item Usage per Thread
- Thread Usage per Item

Select an operation or row in the graph to display it in the Operations or Threads table

below the graph. When you change the selection and focus in a table, the selection and focus also change in the graphs when applicable.

The red portion of an operation indicates the time during which the thread blocks waiting to acquire an item. The green portion of an operation indicates the time during which the thread owns or uses an item. While an operation is blocked, its frame is red. While an operation is in use, its frame is blue. After an operation completes, its frame is black.

Select **Edit»Copy** to copy the entire image of the graph to the clipboard, including the portions of the graph that are not visible because of the scroll position.

The profiler window graphs include the following controls:

- **Zoom %**—Controls the amount of horizontal space a given unit of time occupies on the time line. Increase the value to increase the display size of the operations. Use the horizontal scrollbar when the operations require more horizontal space than is available.
- **Auto Fit**—Select **Auto Fit** to adjust the time scale so all operations fit horizontally within the graph, if possible. Deselect Auto Fit to enable the **Zoom %** control.



Note If a graph has focus, you can zoom in or out using ctrl-mouse-wheel. If the mouse cursor is over the graph, the graph zooms based on the mouse cursor location. You can also choose the Zoom to Selection command from the View menu.

You can drag a bounding box around operations in a graph to select them. You can extend or modify the selection by shift or ctrl clicking on additional operations. The graphs display ruler lines at the start and end of the selection so that you can more easily observe the relative positions of other operations.

Item Usage per Thread Graph

Item Usage per Thread Graph

Use this graph to view the items a thread uses over time. You can view how long the thread waits to acquire an item and how long it retains or uses each item.

Along the vertical axis, each row in the graph represents a TestStand Thread. The thread name displays in a legend to the left of the graph time line. After the thread name, each row displays rectangles that represent the operations in that thread.

The horizontal axis represents time. Operations are positioned along the horizontal axis according to the times at which the operations begin and end. Each operation has a label that indicates the name of the item on which it operates. Hovering over an operation displays a tool tip with additional information.

Thread Usage per Item Graph

Thread Usage per Item Graph

Use this graph to view the threads that use an item over time. You can view how long threads block waiting to acquire an item and how long they retain or use the item.

Along the vertical axis, each row in the graph represents an item. The item name displays in a legend to the left of the graph time line. After the item name, each row displays rectangles that represent the operations on that item.

The horizontal axis represents time. Operations are positioned along the horizontal axis according to the times at which the operations begin and end. Each operation has a label that indicates the name of the thread that performs the operation. Hovering over an operation displays a tool tip with additional information.

Managing Profiler Overhead

Managing Profiler Overhead

The profiler collects information in events that describe the operations it displays. It takes a small amount of time and memory for TestStand to store a profiler event. However, if you profile sequences that execute large numbers of steps that run very quickly and you do not limit the events that the profiler collects, the profiler can collect a number of events large enough to consume a significant amount of time and memory. The number of events the profiler can collect before the effect on performance becomes problematic is system dependent.

If you collect more profiling data than your system can acceptably process, you can do

the following to improve performance:

- **Hide Selected Categories of Data**—You can hide selected categories of data using the options in the View menu. Hiding large numbers of events improves the performance of the profiler window, but is less beneficial than reducing the amount of data collected.
- **Reduce and/or Limit the Amount of Data**—You can control the amount of data the profiler collects using the options in the Configure Data Collection dialog box. This is the most effective option for improving performance.
- **Disable Profiler Display Updates**—You can disable the Update Display button while collecting data. This has a small effect, but can reduce the amount that the operation of the profiler displays impacts the timestamps of the collected events, especially if the profiled operations consume all of the available cpu cores such that the profiler window must share a core with profiled operations.

Database Viewer Application

TestStand includes the Database Viewer application for viewing data in a database, editing table information, and executing SQL commands. You can access the Database Viewer application in the following ways:

- Select **Tools»Database Viewer** in the TestStand Sequence Editor.
- (Windows 8.1) Click the **NI Launcher** tile on the Start screen and select **TestStand»Tools»TestStand Database Viewer**.
- (Windows 10 or 7) Select **Start»All Programs»National Instruments»TestStand»Tools»TestStand Database Viewer**.
- Run `<TestStand>\Components\Tools\DatabaseView\DatabaseView.exe`.
- Run `<TestStand>\Components\Tools\DatabaseView\DatabaseView.exe` from the command line.

The Database Viewer supports the following functionality:

- **Database Explorer**—The Database Explorer pane displays information about open database connections. You can view and edit data, add or remove database tables, and open or close database connections.
- **Database Viewer Tabs**—The Database Viewer contains Edit Data and Execute SQL

tabs, which are associated with an open database connection. All operations performed in each tab execute on the database connection associated with that tab. The connection you select from the connection list when you create a tab becomes the connection associated with that tab.

- **Binary Data Viewer**—The Binary Data Viewer allows you to view binary data that has been logged to the database. You can view the data as a table, a graph, or as the layout in memory.



Note The Database Viewer Application does not require a license for build or analysis purposes, as defined by the License Agreement.

Related concepts:

- [Search Directories](#)
- [Database Logging](#)

Related information:

- [Command Line Arguments](#)
- [Database Viewer Menu Bar](#)

Executing SQL Commands in the SQL Editor Window

Database Viewer supports performing multiple delimited SQL commands in a single operation. The SQL commands you enter are split into multiple commands based on the command delimiter used, and these commands are executed sequentially. The Output Tab is updated with the results of all the commands in the batch.

The various databases management systems supported by TestStand use different delimiters for SQL commands. To maintain compatibility with all of these systems, Database Viewer supports the following command delimiters for databases:

- **Access**— ~
- **MySQL and Sybase**— ;
- **Oracle**— /
- **SQL Server**— GO



Note Access does not support execution of delimited commands. However, you can execute delimited SQL commands using Database Viewer on an Access database.

Use each command delimiter on a new line. You should also add a new line after the delimiter and before the subsequent commands.

You can use any of the command delimiters mentioned above for execution of multiple commands from Database Viewer. Refer to the following example of delimited SQL commands entered using the ; command delimiter for MySQL and Sybase:

```
CREATE TABLE PROP_BINARY
(
ID                                CHAR (38) PRIMARY KEY,
PROP_RESULT                      CHAR (38) NOT NULL,
UPPER_BOUNDS                     VARCHAR (32),
LOWER_BOUNDS                     VARCHAR (32),
DATA_FORMAT                     VARCHAR (32),
DATA                             MEDIUMBLOB,
CONSTRAINT PROP_BINARY_FK FOREIGN KEY (PROP_RESULT) REFERENCES PROP_RESULT (ID)
)
;
CREATE TABLE PROP_NUMERICLIMIT
(
ID                                CHAR (38) PRIMARY KEY,
PROP_RESULT                      CHAR (38) NOT NULL,
COMP_OPERATOR                   VARCHAR (32),
HIGH_LIMIT                     DOUBLE,
LOW_LIMIT                      DOUBLE,
UNITS                           VARCHAR (255),
STATUS                          VARCHAR (255),
CONSTRAINT PROP_NUMERICLIMIT_FK FOREIGN KEY (PROP_RESULT) REFERENCES PROP_RESULT
(ID)
)
;
```

The following individual SQL commands are identified and executed sequentially:

```
CREATE TABLE PROP_BINARY
(
ID                                CHAR (38) PRIMARY KEY,
```

```

PROP_RESULT          CHAR (38) NOT NULL,
UPPER_BOUNDS         VARCHAR (32),
LOWER_BOUNDS         VARCHAR (32),
DATA_FORMAT          VARCHAR (32),
DATA                 MEDIUMBLOB,
CONSTRAINT PROP_BINARY_FK FOREIGN KEY (PROP_RESULT) REFERENCES PROP_RESULT (ID)
)

```

and

```

CREATE TABLE PROP_NUMERICLIMIT
(
ID                CHAR (38) PRIMARY KEY,
PROP_RESULT       CHAR (38) NOT NULL,
COMP_OPERATOR     VARCHAR (32),
HIGH_LIMIT       DOUBLE,
LOW_LIMIT        DOUBLE,
UNITS             VARCHAR (255),
STATUS           VARCHAR (255),
CONSTRAINT PROP_NUMERICLIMIT_FK FOREIGN KEY (PROP_RESULT) REFERENCES PROP_RESULT
(ID)
)
;

```

The SQL commands you enter can also contain comments. For single line comments, use the following syntax:

- **Access, Oracle and SQL Server**— #<comment text><new line>
- **MySQL**— --<comment text><new line>
- **Sybase**— //<comment text><new line>

For multi-line comments, use the following syntax:

```
/*<comment text>
```

```
<comment text>*/
```



Note Single line comments should always start with the comment identifier as the first character in that line.

Customizing the Database Viewer User Interface

You can preview, switch between, dock/undock, float, and group certain tabs and panes in the Database Viewer application.

Document Selector

Press <Ctrl+Tab> to launch the document selector when the Database Viewer window is active to display a list of open panes and tabs and a preview of the selected pane or tab. Hold down the <Ctrl>key and use the arrow keys or the mouse to navigate through the list. The document that is selected when you release the <Ctrl> key becomes the active document when you close the document selector.

Dock/Undock

Tabs in the Database Viewer can be docked/undocked and floated as required. To undock a tab, double-click on the tab or drag it out of the main window or right-click on the tab header and choose the **Float** option.

To dock a floating tab, drag it over the main application window and choose a docking location.



Note

- You can undock and float the Database Explorer pane like a tab. However, this pane cannot be closed.
- Operations selected from the main application menu do not apply to floating tabs. The shortcuts for these operations can be used for the active floating tab to accomplish the same action.

Tab Groups

You can organize tabs into horizontal or vertical groups by dragging the tabs to the location you want or by using the context menu in the tab headers.

Printing and Exporting Data in Database Viewer

You can print and export the data displayed in the data grid of the Edit Data and Execute SQL tabs. You can also print binary data displayed in the tabs of the Binary Data Viewer.

Select **File»Print** or use the <Ctrl+P> keyboard shortcut to launch the Print dialog box, in which you can configure options for printing the data in the currently selected tab.

Select **File»Export Document** to export the data in the currently selected tab and launch the Export Document dialog box, in which you can select the file format for the exported data.

Configuring Connection Strings

A connection string is a string version of the data link connection information that is required to open a database session.

You can use the Data Link Properties dialog box to create a connection string that uses OLE DB providers. You can manually create connection strings that use ODBC Drivers and save them as a `.udl` file. Use these files to specify database connections from TestStand (Database Options, Database Step Types, and Property Loader Step) and from the Database Viewer application.

In the Database Viewer, enter the connection strings in the New Data Link dialog box using the formats given below to establish a connection to the corresponding DBMS servers. Use the **Save Data Link** option to save the connection information as a `.udl` file.

This topic lists the connection string formats for the following DBMS servers and their supported providers:

- Access
- MySQL
- Oracle
- SQL Server
- Sybase

Access

(32-bit TestStand)

OLE DB: Provider=Microsoft.Jet.OLEDB.4.0;Data Source=path to mdb/accdb file;Persist Security Info=False;

- **Example:** Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files\National Instruments\TestStand 2014\Components\Models\TestStandModels\Database\TestStand Results.mdb;Persist Security Info=False;

ODBC: Driver={Microsoft Access Driver (*.mdb)};DBQ=path to mdb/accdb file

- **Example:** Driver={Microsoft Access Driver (*.mdb)};DBQ=C:\Program Files\National Instruments\TestStand 2014\Components\Models\TestStandModels\Database\TestStand Results.mdb

OLE DB: Provider=Microsoft.ACE.OLEDB.12.0;Data Source= path to mdb/accdb file;Persist Security Info=False;

- **Example:** Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\Program Files\National Instruments\TestStand 2014\Components\Models\TestStandModels\Database\TestStand Results.mdb;Persist Security Info=False;

ODBC: Driver={Microsoft Access Driver (*.mdb, *.accdb)};DBQ=path to mdb/accdb file

- **Example:** Driver={Microsoft Access Driver (*.mdb, *.accdb)};DBQ=C:\Program Files\National Instruments\TestStand 2014\Components\Models\TestStandModels\Database\TestStand Results.mdb

MySQL

ODBC: Driver={MySQL ODBC 5.2 ANSI Driver};Server=<database server name>;Database=<schema name>;User=<user name>;Password=<user password>;Option=<options>;

- **Example:** Driver={MySQL ODBC 5.2 ANSI Driver};Server=xfiles2010;Database=TSInsert;User=tester;Password=



Note

- You must install the MySQL ODBC Connector to use MySQL.
- You should specify a default database for the connection. Failure to do so generally results in errors.

Oracle

OLE DB: Provider=OraOLEDB.Oracle.1;Password=<user password>;Persist Security Info=True;User ID=<user name>;Data Source=<data source name as configured in the Oracle client>;Database=<schema name>;

- **Example:**
Provider=OraOLEDB.Oracle.1;Password=teststand;Persist Security Info=True;User ID=tester;Data Source=ORCL;Database=tester;

ODBC: Driver={Oracle in OraClient11g_home1};Dbq=<data source name as configured in the Oracle client>;Uid=<user name>;Pwd=<user password>;

- **Example:** Driver={Oracle in OraClient11g_home1};Dbq=ORCL;Uid=tester;Pwd=teststand;



Note You must install the Oracle Database Client to use the Oracle

providers.

SQL Server

OLE DB:

- **Using SQL Server Native Client:**

```
Provider=SQLNCLI10.1;Password=<user password>;Persist
Security Info=True;User ID=<user name>;Initial
Catalog=<catalog name>;Data Source=<database server
name>;
```

- **Example:** Provider=SQLNCLI10.1;Password=user;Persist
 Security Info=True;User ID=user;Initial
 Catalog=user;Data Source=XFILES2010;

- **Using Microsoft OLE DB Provider for SQL Server:**

```
Provider=SQLOLEDB.1;Persist Security Info=True;User
ID=<user name>;Password=<user password>;Data
Source=<database server name>;Initial Catalog==<catalog
name>;
```

- **Example:** Provider=SQLOLEDB.1;Persist Security
 Info=True;User ID=abc;Password=testing;Data Source=
 XFILES2010\TESTSTANddb;Initial Catalog=test;

- **Using Integrated Security:** Provider=SQLOLEDB.1; Integrated
 Security=SSPI;Data Source=<database server name>;Initial
 Catalog==<catalog name>;



Note This option is available in both the SQL Server Native Client provider and the Microsoft OLE DB Provider for SQL Server. Use this option to create connection strings that do not reveal user authentication information.

ODBC: Driver={SQL Server Native Client 10.0};Uid=<user name>;Pwd=<user password>;Server=<database server name>;Database=<database name>;

- **Example:** Driver={SQL Server Native Client

```
10.0};Uid=user;Pwd=user;Server= XFILES2010\
TESTSTANDDB;Database=test;
```

Sybase

OLE DB: Provider=SAOLEDB.10;ENG=<database engine name>;DBN=<database name>;User Id=<user name>;Password=<user password>;

- **Example:** Provider=SAOLEDB.10;ENG=Diff_Db;DBN=Diff_Db;User Id=diff_tester;Password=sql;

ODBC: Driver={SQL Anywhere 10};UID=<user name>;DatabaseName=<database name>;EngineName=<database engine name>;AutoStop=YES;Integrated=NO;EncryptedPassword=3ff6c3114e;Debug server name>;PORT=<port number for remote connections>};Compress=NO

- **Example:** Driver={SQL Anywhere 10};UID=user;DatabaseName=teststand;EngineName=teststand;AutoSto

Related concepts:

- [Microsoft ADO, OLE DB, and ODBC Database Technologies](#)

Database Viewer Known Issues

The following known issues exist for the Database Viewer application:

- Issues with Using the MSDASQL Provider in Database Viewer

Issues with Using the MSDASQL Provider in Database Viewer

MSDASQL is Microsoft's Object Linking and Embedding Database (OLE DB) provider for Open Database Connectivity (ODBC) drivers. MSDASQL allows OLE DB consumer applications to use ODBC drivers to connect to a data source. The provider converts

incoming OLE DB calls into ODBC calls and passes them on to the specified ODBC driver. The provider then retrieves results from the ODBC driver and formats the results into OLE DB specific structures, which the user can then access.

The Database Viewer application uses the ADO .NET database component, which does not support the MSDASQL provider. If the connection string specified in the New Data Link dialog box uses the MSDASQL provider, the Database Viewer application attempts to remove the MSDASQL provider and connect directly through the ODBC Driver/Data Source Name (DSN) specified in the connection string. If this connection fails, the Database Viewer application will display the following error message:

```
The following error occurred while attempting to connect to
the specified database: The .Net Framework Data Provider
for OLEDB (System.Data.OleDb) does not support the
Microsoft OLE DB Provider for ODBC Drivers (MSDASQL). Use
the .Net Framework Data Provider for ODBC
(System.Data.Odbc). Database Viewer does not support the
MSDASQL provider. For more information on how to resolve
this error, refer to the TestStand help topic : Issues with
Using the MSDASQL Provider in Database Viewer.
```

To resolve this error, you can modify the connection string to connect using the ODBC Driver/DSN directly using one of the two following strategies:

- **Connect using a DSN**—A data source name (DSN) is a data structure that contains information about a specific database that an ODBC driver needs in order to connect to the database. If an existing DSN is to be used to establish a connection with the database, you must simply specify the name of the DSN in the connection string, using the following format: `DSN=MyDSNName`; To create a new DSN, use the Windows ODBC Data Source Administrator tool. (64-bit Windows) Run, with administrator privileges, `[WindowsDir]\SysWOW64\odbcad32.exe` for 64-bit TestStand and `[WindowsDir]\System32\odbcad32.exe` for 32-bit TestStand. (32-bit Windows) Run, with administrator privileges, `[WindowsDir]\System32\odbcad32.exe` for 32-bit TestStand.
- **Connect without using a DSN**—To specify a connection string that directly uses an ODBC driver, refer to the Configuring Connection Strings topic.

Related concepts:

- [Configuring Connection Strings](#)

Related information:

- [New Data Link Dialog Box](#)

Instrument Drivers

Visit the Instrument Driver Network at ni.com/idnet for more information about instrument drivers and for finding and downloading instrument drivers compatible with National Instruments software.

Related information:

- [NI Instrument Driver Network](#)

IVI Drivers

Interchangeable Virtual Instrument (IVI) is an instrument driver standard that provides common programming interfaces for several classes of instruments. IVI drivers exist for a number of National Instruments devices and other popular instruments. Visit ni.com/ivi for more information about IVI. Visit the Instrument Driver Network at ni.com/idnet for more information about instrument drivers and for finding and downloading instrument drivers compatible with National Instruments software.

Two architectures exist for IVI drivers—IVI-C, based on ANSI C, and IVI-COM, based on Microsoft Component Object Model (COM) technology. The IVI step types support IVI-C class-compliant instrument drivers and support IVI-COM class-compliant instrument drivers when you install the IVI-COM Adapter component of the IVI Component Package included in the NI Device Driver DVD. TestStand does not install IVI class instrument drivers.

You can call IVI-C instrument class drivers and specific drivers from any development environment that supports calls into DLLs. Many IVI-C instrument drivers have native LabVIEW-generated wrappers. You can also convert an IVI-C instrument driver using the Create VI Interface to CVI Instrument Driver tool available from the Instrument Driver Network at ni.com/idnet. You can call IVI-COM instrument class drivers and

specific drivers from any development environment that supports ActiveX. Use the ActiveX/COM Adapter to configure steps to access objects IVI-COM class instrument drivers define.

Related concepts:

- [ActiveX/COM Adapter](#)

Related information:

- [Interchangeable Virtual Instrument Drivers](#)
- [NI Instrument Driver Network](#)
- [IVI Step Types](#)

Plug and Play Instrument Drivers

Plug and Play drivers simplify controlling and communicating with the instrument through a standard, straightforward programming model for all drivers. Plug and Play drivers exist for LabVIEW and LabWindows/CVI.

Using Plug and Play Drivers with LabVIEW

A LabVIEW Plug and Play instrument driver is a set of VIs. Each VI corresponds to a programmatic operation for the instrument. National Instruments distributes LabVIEW Plug and Play instrument drivers with the block diagram source code so you can customize the VIs. You can create instrument control applications and systems by programmatically linking instrument driver VIs on the block diagram. LabVIEW Plug and Play instrument drivers usually use Virtual Instrument Software Architecture (VISA) functions to communicate with instruments.

In TestStand, you can call VIs that use LabVIEW Plug and Play instrument drivers. When you return a VISA reference to TestStand and later pass the reference to a different VI code module that uses the same instrument driver, store the reference in a TestStand LabVIEWIORReference variable. You can also use the LabVIEW Adapter to directly call VIs in an instrument driver.

Using Plug and Play Drivers with LabWindows/CVI

A LabWindows/CVI Plug and Play instrument driver is a set of ANSI C software routines exported from a DLL. You can call these instrument drivers from any development environment that supports calls into DLLs. You can also use a LabWindows/CVI instrument driver in LabVIEW when you convert the instrument driver using the Create VI Interface to CVI Instrument Driver tool available from the Instrument Driver Network at ni.com/idnet. LabWindows/CVI Plug and Play instrument drivers are based on the VXI plug & play standard architecture and usually use VISA functions to communicate with instruments.

In TestStand, you can call code modules that use LabWindows/CVI Plug and Play instrument drivers. When you return a C-based reference to TestStand and later pass the reference to a different code module that uses the same instrument driver, store the reference in a TestStand numeric variable. You can also use the LabWindows/CVI or C/C++ DLL Adapter to directly call the functions in an instrument driver.

Related concepts:

- [Code Modules](#)

Related information:

- [NI Instrument Driver Network](#)

Route Specification Strings

When you instruct TestStand to connect or disconnect routes defined in an NI Switch Executive virtual device, you must specify a route specification string. The syntax of a route specification string consists of a series of routes that ampersands (&) delimit. NI Switch Executive ignores white space characters between tokens in a route specification string.

The syntax for a route specification string is as follows:

```
routeOrGroup { & routeOrGroup } { & routeOrGroup } . . .
```

Where routeOrGroup is one of the following:

- Route name
- Route group name
- Fully-specified path—Enclosed in square brackets and consisting of a series of channels that "->" delimits. The following shows the format of a fully-specified path:

```
[ channel {-> channel } {-> channel} . . . ]
```

A channel must be one of the following:

- Channel alias name
- Unique name—A combination of the IVI device logical name and IVI channel name that a "/" delimiter separates
- IVI channel name

Channels on either end of a bracketed, fully specified path must not be a Configuration or a Hardwired channel. Only one end channel can be a Source channel. The inner channels in a route specification string must be either a Configuration or Hardwired channel. The following are examples of route specification strings:

```
[SampleMatrix1/c0->SampleMatrix1/r1->SampleMatrix1/c4]
```

```
[Scope->R3->SampleMatrix1/c6]
```

```
ArbToInput & ScopeToOutput
```

```
PowerDevice & [Scope->R3->UUT_Out]
```

To specify a route specification string in an expression, the expression must return a valid route specification string when TestStand evaluates the expression. The following are examples of expressions that return valid route specification strings:

```
"PowerDevice & [Scope->R3->UUT_Out]"
```

```
Locals.MyRouteGroupName + "& ScopeToOutput"
```

```
SwitchFindRoute("DeviceName", "DeviceChan1", "DeviceChan2")  
+ "&" + Locals.RouteName
```



Note

- You can use NI Switch Executive to find an available path at run time by slightly modifying the syntax. Remove the square brackets and replace them with the two endpoints of the route that a `->` delimits.
- Any channel, other than an endpoint, within a route specification string must be reserved for routing or directly hardwired to one of the endpoint channels. Channels, other than endpoints, must not be configuration channels.
- When connecting a route, the list of channels may only include a single source channel either explicitly or implicitly by means of a previous connection. Attempts to connect two sources together will result in an error.
- You can use the `SwitchFindRoute` expression function to dynamically determine a physical route path at run time.

Related concepts:

- [Expressions](#)

Sequence File Converter

Use the Sequence File Converter tool to resave sequence files on disk using the current version of TestStand. You can also change the file format to binary, XML, or INI, and you can change the protection setting for the files. You can launch the tool in the following ways:

- Select **Tools»Update Sequence Files** in the TestStand Sequence Editor or a in full-featured TestStand User Interface.
- (Windows 8.1) Click the **NI Launcher** tile on the Start screen and select **TestStand»Tools»TestStand Sequence File Converter**.
- (Windows 10 or 7) Select **Start»All Programs»National Instruments»TestStand»Tools»TestStand Sequence File Converter**.
- Run `<TestStand>\Bin\SequenceFileConverter.exe`.

The application window displays a view of the directories and files on disk. You can

change the base path of the view by entering a path in the toolbar, by selecting a directory using the **Browse** button, or by double-clicking on folders in the view.

Use the Edit menu and the context menu to change the file format and protection setting for the files you select in the view. Setting changes made to sequence files in the sequence file converter application are not written to disk until you select **Edit»Apply Changes**, click **Apply Changes** in the toolbar, or select **Update Files** from the context menu.

Command-Line Usage

```
SequenceFileConverter.exe [-Help] [-Recursive] [-Quiet] [-UpdateVersion] [-Unlock:<PasswordList>] [-Format:<Format>] [-Protect:<Mode>:<Password>] <file or directory path> [-env <environment path>]
```

Example

```
SequenceFileConverter.exe -Recursive
-Unclock:myPass,myOtherPass -Format:Binary
-Protect:View:myPass "C:\My Test Directory"
```

This example shows how to complete the following tasks:

- Unlock all files under "C:\My Test Directory" using either "myPass" or "myOtherPass".
- Convert unlocked files to binary format.
- Set password protection to prevent viewing of the converted files, using "myPass".



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

Command-Line Options

The following list describes all of the available options you can use in the command line for the sequence file converter:

- **Help**—Launches a dialog box that contains an explanation of the command-line arguments.
- **Recursive**—Applies changes to files found in directories below the specified directory.
- **Quiet**—Disables the prompt to confirm conversion and prompts to unlock files.
- **UpdateVersion**—Saves sequence files with the current version of TestStand even if no other modifications to the file exist.
- **Unlock:<PasswordList>**—Attempts to unlock files for editing using a comma delimited password list.



Note This does not remove password protection.

- **Format:<Format>**—Converts all files to the specified format. Possible values for <Format> are binary, INI, and XML.
- **Protect:<Mode>:<Password>**—Protects all files with <password>. Possible values for <Mode> are Edit, View, and Disable.



Note The Disable value removes password protection from all files unlocked during conversion.

- **env<environment path>**—Launches the Sequence File Converter tool in the environment defined by the `.tsenv` file located at the path you specify. Consult *Defining an Environment* for more information about `.tsenv` files and *Choosing an Environment* for more information about choosing an environment with the `-env` command line switch.

Related concepts:

- [Search Directories](#)
- [Defining an Environment](#)
- [Choosing an Environment](#)

TestStand Version Selector

The TestStand Version Selector application indicates the active version of TestStand, can activate a different version of TestStand, and can reconfigure the currently active version. Although you can install more than one version of TestStand on a computer, only one version of TestStand can be active and can run at a time. You can launch the application in the following ways:

- (Windows 10) Click **Start»National Instruments»TestStand Version Selector**.
- Run `C:\<Program Files>\National Instruments\Shared\TestStand Version Selector\TSVerSelect.exe`.

The TestStand Version Selector initiates the following actions for all versions of TestStand:

- Registers the ActiveX interfaces for the TestStand API.
- Registers the ActiveX interfaces for the TestStand User Interface (UI) Controls.
- Registers the TestStand API and UI Controls interop assemblies for Microsoft Visual Studio.
- Registers various ActiveX controls, DLLs, and applications located in the `<TestStand>\Components` directories.
- Adds the TestStand VIs and Functions Palette and the TestStand Controls Palette to installed versions of LabVIEW.

The TestStand Version Selector also initiates the following actions for specific versions of TestStand:

Version Activated	Actions
TestStand 2021 SP1 or later	Installs NI TestStand Support for Debugging Python Code, which is an extension for Visual Studio Code. This action occurs only when VS Code is already installed on the computer. When you select TestStand 2021 SP1 or later under Installed TestStand versions, and then click either Activate or Configure, the TestStand Version Selector installs the VS Code extension. If you want to perform step into debugging of Python modules, you must complete this step to install the VS Code extension.

Version Activated	Actions
TestStand 4.1 or later	• Copies API files to <National Instruments>\Shared\CVI\instr\TestStand\API for LabWindows/CVI 8.5 or later, or to <CVI>\instr\TestStand\API for LabWindows/CVI 8.1.1 or earlier. • Adds environment variables for the <TestStand>, <TestStand>\Bin, <TestStand Public>, and <TestStand Application Data> directories. You can use the environment variables in development environments to specify paths to files in these TestStand directories. For example, you can use these environment variables to build paths in the Additional Include Directories control in Visual Studio.  Note The environment variables for 64-bit TestStand end with a 64 suffix. For example, 32-bit TestStand uses the variables TestStand, TestStandBin, TestStandPublic, and TestStandAppData, and 64-bit TestStand uses the variables TestStand64, TestStandBin64, TestStandPublic64, and TestStandAppData64. In TestStand 4.1 and later, the Current Active Version control shows the currently active version of TestStand. Select a version in the Installed TestStand versions control and click the Activate button to activate a specific version of TestStand. The Path to 32-bit version and Path to 64-bit version controls show the location of the selected version of TestStand. When activation is complete, the TestStand Version Selector indicates success or displays any warnings. Click the Launch 32-bit or Launch 64-bit button to run the sequence editor application for the active version of TestStand.  Note The TestStand Version Selector does not normally require administrator privileges to activate a specific version of TestStand on Microsoft Windows 10. To require administrator privileges for using the TestStand Version Selector, add the <code>/reqadmin</code> command-line option to the selector shortcut.
TestStand 4.0.1 or earlier	Adds the <TestStand>\API\CVI\ directory to the LabWindows/CVI development environment search directories for include files and instrument drivers. You must exit LabWindows/CVI before activating a new version of TestStand to ensure the changes take effect.

Command-Line Options

The TestStand Version Selector supports the following command-line options:

Argument	Description
/?	Displays help for command-line options.
/version <versionname>	Activates specified version, where <versionname> is the version of TestStand to activate.
/launch <filename>	Launches the executable you specify, where <filename> is the path to the executable to launch.
/noprompt	Suppresses prompts while performing activation and launch operations.
/reqadmin	Forces the Version Selector to require Administrator access when performing activation.
/installing	Displays a status prompt while activating a specified version without displaying errors. You must use this option with the /version and /noprompt options.
/donotreactivate	Do not reactivate TestStand if the version of TestStand is already active.

Log File

The TestStand Version Selector creates log files, located at <CommonAppData>\National Instruments\TestStand Version Selector, when you run the version selector to activate a version of TestStand and when you run an application that uses the TestStand Engine because the engine uses the version selector to update external environments. The log files contain details about the changes the version selector made, which might be useful when debugging issues related to changing the active version of TestStand.

Related concepts:

- [Search Directories](#)
- [Activating 32-bit TestStand and 64-bit TestStand](#)

Related information:

- [TestStand Version Selector Environment Reference](#)

Using the TestStand Version Selector

Although you can install more than one version of TestStand on a computer, only one version of TestStand can be active and run at a time. The TestStand Version Selector application indicates the active version of TestStand and can activate a different version of TestStand.

The TestStand Version Selector application, `TSVerSelect.exe`, is located in the `<Program Files>\National Instruments\Shared\TestStand Version Selector` directory. To launch the TestStand Version Selector, select **Start » NI » TestStand Version Selector**.



Note The TestStand Version Selector does not typically require administrator privileges to activate a specific version of TestStand on Microsoft Windows 10. To require administrator privileges for using the TestStand Version Selector, add the `/reqadmin` command-line option to the selector shortcut.

The **Current Active Version** control shows the currently active version of TestStand. Select a version in the **Installed TestStand Versions** control and click **Activate** to activate a specific version of TestStand. The **Path to 32-bit Version** and **Path to 64-bit Version** controls show the locations of the selected version of TestStand.

When activation is complete, the selector displays a message to indicate success or display any warnings. Click **Launch 32-bit** or **Launch 64-bit** to run the sequence editor application for the active version and bitness of TestStand.

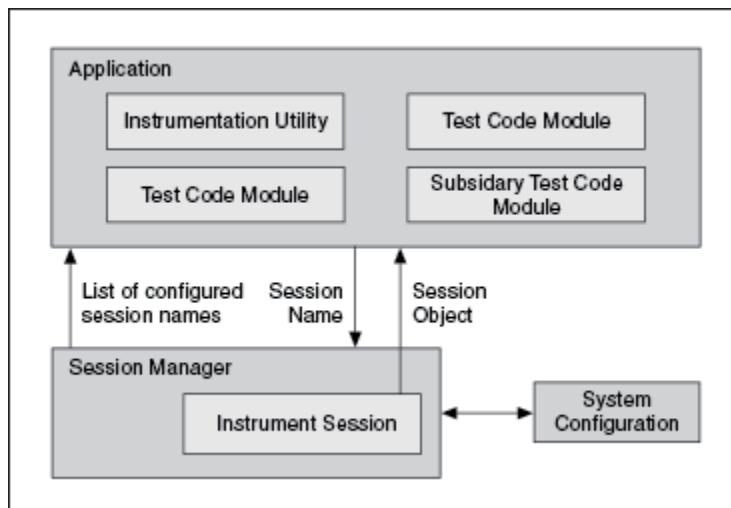
If you activate TestStand and run a TestStand User Interface from a previous TestStand version, the user interface uses the TestStand Engine, step types, and components for the active TestStand version. User interfaces and sequence editors created with older versions of TestStand may not run as expected if a newer version of TestStand is active.

Session Manager

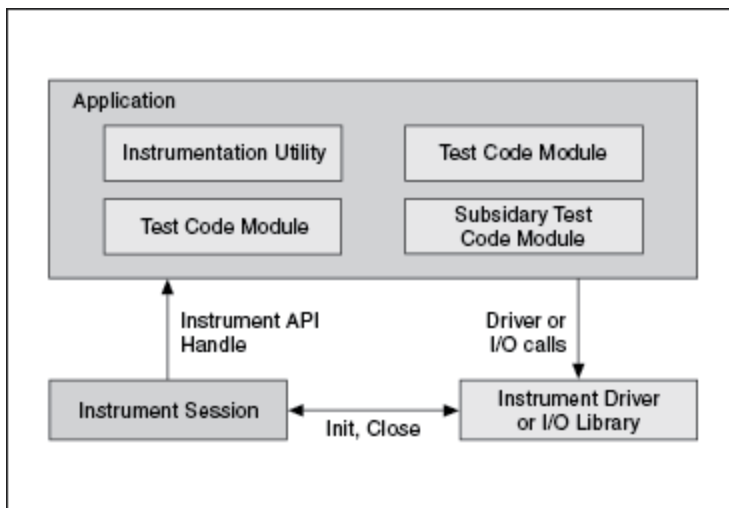
Use Session Manager to obtain instrument sessions. Session Manager returns a reference to the session object with the name you specify. If the session object you specify does not yet exist, Session Manager creates it and returns a reference to the new session. You can also query the names and categories of available sessions.

Session Manager is a single instance, in-process server that is globally available to all software components in the process.

Various types of modules in an application can use Session Manager to obtain instrument session objects and configuration information, as shown in the following figure.



Application modules can also use an instrument session to obtain instrument API handles, as shown in the following figure.



Instrument Sessions

Instrument sessions are ActiveX objects that enable software modules to share one of the following types of instrument API handles easily:

- An instance handle for an instrument driver, such as an IVI-C or VXIplug&play driver
- An ActiveX reference for an instrument driver, such as an IVI-C driver



Note Session Manager does not support IVI-COM drivers.

- An instance handle or ActiveX reference for an I/O library object, such as a VISA session
- An instance handle for a virtual device the NI Switch Executive defines

You can use an instrument session to obtain a single instrument API handle or the same API handle multiple times in different software components. The instrument session initializes the API handle when you first request the handle and closes the handle when you release the last reference to the session. Typically, you do not explicitly initialize or close the instrument API handle.

During configuration, you assign each instrument session a unique name. To reference an instrument session, specify the unique name of the session.

Why Use Instrument Sessions?

Instrument sessions are useful when you must share instrument API handles within a single process among loosely coupled software components. A test management system is an example of a system that can benefit from instrument sessions.

Using named instrument sessions can include the following advantages:

- **Reduced Coupling**—You do not have to pass instrument handle variables among software components. Instead, each component specifies a logical instrument name to obtain a session. Components then acquire the API handle from the session. Because you specify a logical instrument name to access a session, components that are not explicitly connected to each other can access the same session and thereby share an API handle. Because the session initializes and closes the API handle as needed, components that use the session do not need to be aware of the relative order in which they execute.
- **Reduced Programming Language Barriers**—Code modules written in different languages can share the same session without passing handles that might not easily convert among languages.
- **Lifetime Control**—Because instrument sessions are ActiveX objects with reference counts, you can tie the lifetime of the session to the lifetime of an ActiveX reference variable and eliminate the need to explicitly close the instrument in languages that support ActiveX reference variables. Multiple variables can reference the same session. The session closes when its last reference releases. Because a session lives as long as a reference to it exists, you do not have to coordinate the lifetime of the session among multiple threads or processes that use it.
- **Efficiency and Correctness**—A session initializes the API handle only when necessary. For example, if a component uses a logical instrument name and calls a subcomponent that uses the same logical name, the subcomponent can share the session that its caller initializes. Sharing the session avoids the overhead of initializing a second API handle. By sharing a session, you also avoid the possibility of using a second API handle with the same instrument resource, which might invalidate the internal state of the first API handle.
- **Virtual Instrument Selection**—You can query Session Manager to obtain a list of logical and/or virtual instrument names from which to make a selection. You can also narrow the query to a subset of the available session names. For example, you might make one of the following queries:

- All virtual instrument and/or logical names for sessions that provide an instrument API handle to an IVI Switch class driver.
- All virtual instrument and/or logical names for sessions that provide an instrument API handle for a specific VXIplug&play driver.



Note Although instrument sessions are ActiveX objects, the underlying instrument handles, specifically VISA, VXIplug&play, IVI C, and NI Switch Executive, are not valid across processes. For example, you cannot use an IVI C-based handle that a step in the TestStand process creates in a VI code module that executes using the LabVIEW development system process.

Instrument Session Types

Session Manager supports the following instrument session types:

- **IVI Sessions**—Use an IVI session to obtain the C-based instance handle for an IVI logical or virtual instrument name.



Note NI Session Manager does not support IVI-COM drivers.

- **VXIplug&play Sessions**—Use a VXIplug&play session to obtain a C-based instance handle for a VXIplug&play logical or virtual instrument name. Configure VXIplug&play names in the `NI_SessionMgr.ini` file located in the `<VXIplug&play directory>\WinNT or Win64 directory` (for example, `C:\<Program Files>\IVI Foundation\VISA\WinNT\NI_SessionMgr.ini`).
- **VISA Sessions**—Use a VISA instrument session to obtain a C-based `viSession` handle for a VISA resource or logical name. Configure VISA logical names in the `NI_SessionMgr.ini` file located in the `<VXIplug&play directory>\WinNT or Win64 directory` (for example, `C:\<Program Files>\IVI Foundation\VISA\WinNT\NI_SessionMgr.ini`).
- **Custom Sessions**—Use a custom session to create a data container object that shares ActiveX objects you create or other data among software components you write. Use the `Attach` and `Get` methods to attach data to and retrieve data from a session. A custom session does not initialize, close, or own an instrument handle. The data you share with a custom session does not have to be instrumentation

related. You can create a custom session with any name you request.



Note Always prefix custom session names with `CUSTOM: : .`

- **SE Sessions**—Use an NI Switch Executive session to obtain the C-based instance handle for a virtual device.



Note The handles you obtain for IVI, VXIplug&play, and VISA are occasionally referred to as sessions. The instrument sessions Session Manager creates are smart sessions that manage IVI, VXIplug&play, and VISA sessions.

Instrument Session Names

Each instrument session has one or more names. You can use any of the names to obtain an instrument session reference from Session Manager. The names consist of the following elements:

- **IVI or VXIplug&play session**—Virtual instrument name and any number of logical names. A VISA session has a resource name and any number of logical names.
- **IVI or VXIplug&play virtual instrument name**—Refers to the combination of a software driver, a hardware resource, and a set of configuration options.
- **IVI or VXIplug&play logical name**—Alias for a particular IVI or VXIplug&play virtual instrument name.
- **VISA resource name**—Identifies a serial port, GPIB address, or other hardware resource. A VISA logical name is an alias for a particular VISA resource name.
- **Switch Executive Virtual Device**—Refers to an NI Switch Executive virtual instrument name defined in Measurement & Automation Explorer (MAX).



Note Session Manager does not support IVI-COM drivers. You must instead create the IVI-COM ActiveX reference and access the IVI-COM interface natively within the development environment.

Example Instrument Session Names with Categories:

IVI::SampleScope	IVI logical name

IVI::Hpe1463a	IVI virtual instrument name
VXIIPNP::Digital Test Instrument 1	VXIplug&play logical name
VXIIPNP::Term9	VXIplug&play virtual instrument name
VISA::Serial Port 1	VISA logical name
VISA::ASRL1::INSTR	VISA resource name



Note If the resource descriptor is an IVI name, you must match the case-sensitive entry in the IVI Configuration Store. All other instrument session names are case insensitive.

Instrument Session Categories

Each type of instrument session has a category name. You can prefix a session name with a category to distinguish sessions of different types that might have the same name. Separate the category from the session name with any of the following separators: ::, \, \\, \/, /, //, /\, :, :\\, :/, ://, :/\, :\\/, :\\..

If you query Session Manager for the available logical names and specify that you want Session Manager to prefix the categories to the names, Session Manager uses :: as the category separator.

Session types can have subcategories, but only IVI sessions support subcategories. The IVI session subcategories are the IVI class prefixes. You do not have to specify the IVI subcategory in a logical name. However, when querying for logical names, you might want to specify the subcategory to restrict the set of names Session Manager returns to those that represent instruments of a specific IVI class.

Example Categories

- Custom
- IVI
- IVI::IviDMM
- IviDMM
- SE

- VXIPNP
- VISA

Example Instrument Session Names with Categories

- Custom::MyInstrument
- IVI::SampleDMM
- IVI::IviDMM::SampleDMM
- IviDMM::SampleDMM
- SE::SwitchExecutiveExample
- VXIPNP::LCLSA
- VISA::ASLR1::INSTR



Note If the resource descriptor is an IVI name, you must match the case-sensitive entry in the IVI Configuration Store. All other instrument session names are case insensitive.

Related concepts:

- [Instrument Session Types](#)
- [Instrument Session Names](#)

Instrument Session Name Configuration

The location in which you configure instrument session names depends on the type of session. For IVI sessions, configure the logical and virtual instrument names using a utility, such as Measurement & Automation Explorer (MAX).

You can use all the IVI aliases for a session, including logical names and the virtual instrument name, with or without the `VInstr` prefix. For example, for the same session, you might use `SampleDMM`, `VInstr->FL45`, or `FL45`.

Session Manager maintains VXIplug&play and VISA configuration information in the `NISessionMgr.ini` file, located in the `<VXIplug&play directory>\WinNT` or `Win64` directory (for example, `C:\<Program Files>\IVI Foundation\VISA\WinNT\NISessionMgr.ini`). The `NISessionMgr.ini` file defines the VXIplug&play logical and virtual instrument names and the VISA logical names.

Session Manager calls the VISA function `viFindRsrc ("?*INSTR")` to obtain the list of VISA resource names.

If you change the `NISessionMgr.ini` file or the IVI settings in MAX, Session Manager loads the changes the next time you query for the available session names or request a new session object. The settings of an existing session object do not change when you alter a configuration file. If you release all references to a session and then request the same session, Session Manager recreates the session with the new configuration settings.

NISessionMgr.ini Configuration File

`NISessionMgr.ini` is a session manager, virtual instrument, or configuration file that you use to associate a hardware resource with a VXIplug&play driver and assign a virtual instrument name to the combination of the hardware resource and VXIplug&play driver. You can also use this file to assign logical names to VXIplug&play virtual instruments and VISA resources.

If the `NISessionMgr.ini` file does not exist, Session Manager automatically regenerates it by scanning the contents of the `IVI Foundation\VISA` directory to create a default entry for each installed VXIplug&play driver. The generated file uses the following format:

[VXI PNP Logical Names]

<logical name> = <virtual instrument name>

<another logical name> = <another virtual instrument name>

[VISA Logical Names]

<logical name> = <VISA resource name>

<another logical name> = <another VISA resource name>

Session Manager ignores VXIplug&play virtual instruments that do not specify all the following fields:

[VPPVInstr-><virtual instrument name>]

ModulePath = <dll path>

Prefix = <instrument prefix>

Resource = <VISA hardware resource>

IDQuery = <True or False>

Reset = <True or False>

The following list shows examples of VXIplug&play virtual instruments that might occur in the generated file:

[VXI PNP Logical Names]

DMM1 = "VPPVInstr->FlukeDMM"

[VISA Logical Names]

Serial1 = "ASRL1::INSTR"

[VPPVInstr->FlukeDMM]

ModulePath = "C:\Program Files\IVI Foundation\VISA\WinNT\Bin\fl45_32.dll"

Prefix = "FL45"

Resource = "GPIB::14::INSTR"

IDQuery = True

Reset = True

When Session Manager generates the `NISessionMgr.ini` file, each VXIplug&play virtual instrument resource tag is an empty string. You must edit the file to specify the resource for each virtual instrument. Session Manager ignores virtual instruments that do not specify a resource.

Use the following guidelines to edit the `NISessionMgr.ini` configuration file:

- Each virtual instrument name defaults to the instrument prefix.
- Edit the section name to rename a virtual instrument.
- You can copy existing sections to create new virtual instruments.
- Session Manager does not create any default logical names.



Note Session Manager obtains configuration information about IVI drivers from MAX. By default, MAX does not specify whether to perform an ID query or a reset operation when initializing a virtual instrument. If you do not specify otherwise, Session Manager always performs the ID query and reset operation. You can add entries to the `NISessionMgr.ini` file to explicitly specify whether Session Manager performs these operations. To specify the operation settings in the `NISessionMgr.ini` file, add a virtual instrument definition section for the IVI virtual instrument, as shown in the following example:

```
[VInstr->FL45]

Reset = True

IDQuery = False
```

Refer to the `NISessionMgr.ini` file in the `IVI Foundation\VISA\WinNT` or `Win64` directory for more information about the contents of this file.

Related concepts:

- [Instrument Session Types](#)

VXIplug&play Base Interface

You can use VXIplug&play and IVI instrument sessions to obtain an ActiveX reference to a VXIplug&play base interface. You can use this interface to call the following required VXIplug&play non-initialization methods without calling particular driver APIs:

- `VPPBase.Error_Query`
- `VPPBase.Error_Message`

- VPPBase.Revision_Query
- VPPBase.Self_Test
- VPPBase.Reset

For example, you can use the VXIplug&play base interface to create a utility that resets or self-tests all instruments in the system.

Related information:

- [VPPBase API Reference](#)

Limitations with Sharing Instrument Sessions Among Processes

Any software component in any thread of a process can use Session Manager to discover and access instrument sessions that other components in the same process use. Instance handles that an instrument session creates are C-based API handles. Attempting to share C-based VISA, VXIplug&play, IVI C, or NI Switch Executive instance handles across processes results in an error. For example, you cannot use an IVI C-based handle that a step in the TestStand process creates in a VI code module that executes using the LabVIEW development system process.

By default, Session Manager does not help you access a session that a component in a different process uses. However, you can create a cross-process session that any process can discover and access through Session Manager. To create or reference a cross-process session, prefix an asterisk character to the session name. For example, different processes might share a custom session by specifying

*Custom::MyCustomName. Because Session Manager instantiates cross-process sessions in a separate, shared server process, any C-based API instance handles that a session creates are not valid in any client process.



Note The only session category for which you might consider creating a cross-process session is the Custom category.

All Development Environments

Any component that requires direct access to Session Manager can obtain a reference

to Session Manager by creating a session manager object. Because Session Manager is a single instance, in-process ActiveX server, each component in the process receives a reference to the same session manager object. After you use Session Manager to obtain a session or perform a query, you can release the session and then recreate it later when you need it again.

The base name of the Session Manager server is `SessionMgr`. For ActiveX environments in which you cannot browse for a registered server, you can import the type information you need to access the Session Manager server from `NI_SessionServer.dll`.

Using Instrument Sessions in a Development Environment

You can use instrument sessions in the following ways:

- Use instrument sessions implicitly by using an ActiveX control, a TestStand step type, or another high-level component that uses instrument sessions to facilitate instrument communication.
- Call Session Manager to obtain an instrument API handle you use in a code module.
- Directly call Session Manager to implement components or utilities that use more advanced Session Manager features, such as enumerating available session names or locking sessions.

You can use an instrument session in the following development environments:

- LabWindows/CVI
- LabVIEW
- Microsoft Visual C++ and Other C++ Environments
- TestStand

Related concepts:

- [LabWindows/CVI](#)
- [LabVIEW](#)
- [Microsoft Visual C++ and Other C++ Environments](#)
- [TestStand](#)

LabWindows/CVI

TestStand includes a LabWindows™/CVI™ instrument driver, `SessionMgr.fp`, that provides a complete LabWindows/CVI ActiveX wrapper for the Session Manager server. The functions in the Advanced class of the driver contain the following additional functions to further simplify obtaining instrument API handles in test code:

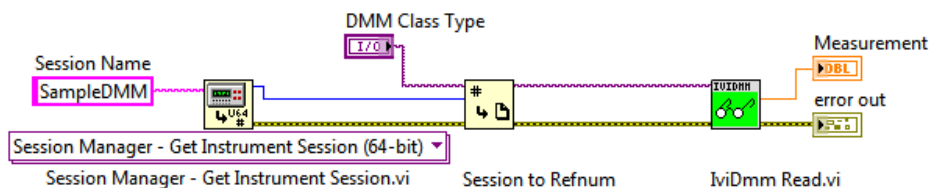
- `SM_GetSession`
- `SM_GetHandle`
- `SM_ReleaseSession`

The instrument help for the `SessionMgr.fp` provides an example of how to use these functions in a code module.

LabVIEW

You can use LabVIEW ActiveX Invoke and Property nodes to access the complete functionality of the Session Manager server.

The following figure demonstrates how to use the TestStand Session Manager - Get Instrument Session VI to obtain an instrument API handle:



Note Instance handles that an instrument session creates are C-based API handles. Attempting to share C-based VISA, VXIplug&play, IVI C, or NI Switch Executive instance handles across processes results in an error. For example, you cannot use an IVI C-based handle that a step in the TestStand process creates in a VI code module that executes using the LabVIEW development system process.

Microsoft Visual C++ and Other C++ Environments

Microsoft Visual C++ code modules can call the LabWindows/CVI instrument driver,

`SessionMgr.fp`, or use the compiler ActiveX support to call the Session Manager server directly.

TestStand

You implicitly use instrument sessions in TestStand when you use IVI step types. IVI step types can share sessions with code modules written in any language that supports ActiveX.

You can also use the ActiveX/COM Adapter to access the complete functionality of the Session Manager server from ActiveX steps in a sequence. Complete the following steps to call Session Manager from an ActiveX step.

1. Right-click the step and select **Specify Module** from the context menu.
2. Select **NI Session Manager** from the **Automation Server** ring control.



Note Instance handles that an instrument session creates are C-based API handles. Attempting to share C-based VISA, VXIplug&play, IVI C, or NI Switch Executive instance handles across processes results in an error. For example, you cannot use an IVI C-based handle that a step in the TestStand process creates in a VI code module that executes using the LabVIEW development system process.

Initialize with Options

An option string configures the IVI driver features of state caching, range checking, simulation, status checking, and record value coercions. For instrument drivers that support a family of instruments, you can use the `DriverSetup` option string to set the particular model of instrument you want the driver to emulate. The IVI Engine then uses the option string settings when it runs the test application.

To enable an IVI feature, set the value to `1` in the option string. To disable a feature, set the value to `0`. You do not have to specify all the properties. If you do not specify one of the properties, the session uses the default value. If you pass `NULL` or an empty string for the options parameter, the session uses the default values for the properties.

The following table lists IVI driver features, the corresponding strings for the option

string, and the default value setting for each feature. The IVI Engine uses these default values when you do not name a feature in the option string. A comma separates each option you specify.

Feature Option	String	Default Value	Default State
State Caching	Cache	1	Enabled
Range Checking	RangeCheck	1	Enabled
Simulation	Simulate	0	Disabled
Status Checking	QueryInstrStatus	1	Enabled
Record Value Coercions	RecordCoercions	0	Disabled
NI I/O Trace NI I/O Trace is an application that monitors, records, and displays National Instruments API calls made by applications.	Spy	0	Disabled
Interchangeability Checking Interchangeability checking verifies that the program produces the same behavior when you use it with a different instrument.	InterchangeCheck	0	Disabled
Specific Instrument Control Use the DriverSetup option to specify a specific model of an instrument you want to control.	DriverSetup	""	""

For example, the option string: `"Simulate=0, RangeCheck=1, QueryInstrStatus=1, Cache=1, Spy=1, InterchangeCheck=0, DriverSetup = Model:6652A"` results in the following configuration:

- Instrument simulation is disabled
- Range checking is enabled
- Status checking is enabled
- State caching is enabled
- Recording of coercions of values is disabled
- NI I/O Trace is enabled
- InterchangeCheck is disabled
- Control of the HP 6652A model

Refer to the IVI Driver Help for more information about IVI driver feature options.

Session Locking

Lock a session before you acquire and use its instrument API handle when you want to prevent another thread or process from simultaneously using the same instrument handle. Use the following methods to implement session locking:

- `InstrSession.GetLockState`
- `InstrSession.LockSession`
- `InstrSession.UnlockSession`
- `InstrSessionMgr.GetUniqueKeyName`

Session locking is effective only when all components that access the session use the locking functions properly. Locking a session blocks other threads or processes that lock the session before they use it. Locking a session in one component cannot prevent another component that fails to call the locking functions from caching the instrument handle and using the instrument API on that handle. In this way, a session lock is like an operating system critical section or mutex you use to control access to a shared resource.

Unlike an operating system mutex, however, the thread that calls the `InstrSession.LockSession` method does not own the lock. Instead, the key string you pass to the method owns the lock. Pass the Thread ID or Process ID as the key string to lock on a per-thread or per-process basis. Pass a string that uniquely identifies the object that owns the lock, such as `TestFixture1`, to lock on an object basis. Pass a key string that you obtain by calling the `InstrSessionMgr.GetUniqueKeyName` method to lock on an absolute basis.

If you call the `InstrSession.LockSession` method on a session that is already locked with a different key, the calling thread sleeps until you acquire the lock or until the timeout period you specify expires.

Attaching Additional Data to a Session

Use the following methods to attach an object or data structure to a session object. You can also attach additional objects to an instrument session to implement a component that represents the instrument connection at a higher level of abstraction.

- InstrSession.AttachLong
- InstrSession.AttachLongPtr
- InstrSession.AttachObject
- InstrSession.AttachString
- InstrSession.DetachData
- InstrSession.GenerateUniqueID
- InstrSession.GetLong
- InstrSession.GetLongPtr
- InstrSession.GetObject
- InstrSession.GetString

What Software Do I Install to Use Instrument Sessions?

You must install the Session Manager ActiveX server to use instrument sessions. The Session Manager ActiveX server requires NI-VISA but does not require LabVIEW, TestStand, LabWindows/CVI, the LabWindows/CVI Run-Time Engine, or IVI. As an ActiveX server, Session Manager runs only on Microsoft Windows platforms.

TestStand Migration Utility

Use the TestStand Migration Utility to copy settings, configuration, and public component files from the folders of previous versions of TestStand to the current version of TestStand. With TestStand 2014 or later, you can also migrate from 32-bit TestStand to 64-bit TestStand or from 64-bit TestStand to 32-bit TestStand. National Instrument recommends migrating between bitnesses only within the same version of TestStand.

You can launch the utility in the following ways:

- (Windows 7) Select **Start»All Programs»National Instruments»TestStand»Tools»Compatibility»TestStand Migration Utility**.
- (Windows 8.1) Click the **NI Launcher** tile on the Start screen and select **TestStand»Tools»Compatibility»TestStand Migration Utility**.
- (Windows 10) Select **Start»National Instruments»TestStand Migration Utility**.

Use the Files to Copy section to specify the files you want to copy. By default, the migration utility selects files you added or modified in the previous version of

TestStand. Expand the directories in the Files to Copy control to view the files you can migrate and determine whether those files have changed since installation. The Conflicts column includes any conflicts that could affect the migration of the files you specify.

The Files to Copy section includes three columns.

- **Files to Migrate**—Expand the directories to view and select the files to migrate to the current version of TestStand. When you enable a file for migration, the Migration Action column updates to display the necessary action.
- **Since Install**—Indicates whether a file has changed or has been added since you installed TestStand 2012 or later.
- **Migration Action**—Indicates whether the Migration Utility copies the files to a folder in the current version of TestStand or overwrites an existing file in a folder in the current version of TestStand. The utility adds the contents of a copied folder to the contents of a folder in the current version of TestStand if a corresponding folder already exists. The Migration Action column displays the following actions:
 - **Copy**—The file does not exist in the previous version of TestStand.
 - **Overwrite**—The file exists in the previous version of TestStand. The migration utility will overwrite the existing file.
 - **Overwrite readonly**—The file exists in the previous version of TestStand but is read-only. The migration utility will not overwrite the existing file.

The Migration Action column is empty for files not selected for migration.

Enable the **Hide Files not modified since installing TestStand** option to show only files modified or created in the version of TestStand you are migrating from.

Use the **Location for backup of TestStand public files** control to specify the location where the Migration Utility stores the backup copy of the files before performing the copy.

When the migration completes successfully, the migration utility generates a text file report that contains the results of the migration, including where the TestStand migration utility stored the file backups and which files were moved. If the migration utility moved result processing configuration files, the migration report specifies the migrated configurations and the name of the new configuration set.



Note

- This utility overwrites any file in the current version of TestStand if the file also exists in the corresponding folder for the previous version of TestStand and if the file is selected in the **Files to Copy** control. Although the Migration Utility creates a backup copy of all overwritten files, National Instruments recommends that you verify that the utility will not overwrite any files in the current version of TestStand that you intend to keep.
- If the `<TestStand Public>\Components` directory of the previous version of TestStand includes files installed by an installer other than the TestStand installer, you might need to use that specific installer to move the files for the files to function and uninstall properly from the `<TestStand Public>\Components` directory of the current version of TestStand. National Instruments recommends that you use an installer that targets the current version of TestStand to install any such components instead of using the migration utility to copy the components. However, some installed components can function properly when copied to another directory and are safe to migrate with the migration utility. The migration utility only moves files and does not update any other information or settings, such as registry keys.
- The utility does not copy files in the `<TestStand Public>\Examples`, `<TestStand Public>\Tutorial`, or `<TestStand Public>\AdapterSupport` directories. These directories do not appear in the Files to Migrate column of the utility.

Cross-Bitness Migration

National Instruments recommends upgrading to 64-bit TestStand 2014 in two steps. First upgrade to 32-bit TestStand 2014 and validate that the system works as expected, and then migrate from 32-bit TestStand 2014 to 64-bit TestStand 2014. Performing the upgrade in two steps can help isolate the root cause of issues you might encounter.

Migrating to 64-bit TestStand might require additional manual steps after the Migration Utility completes. You must update bitness-specific files to support the new

architecture. For example, if the Migration Utility copies a LabWindows/CVI DLL code module from 32-bit TestStand to 64-bit TestStand, 64-bit TestStand cannot load the 32-bit DLL. You must replace the 32-bit DLL with an equivalent 64-bit DLL. Additionally, code modules you place or install under the `<TestStand Public>` directory might require additional work depending on the type and implementation of the code module.

Related concepts:

- [Search Directories](#)
- [64-bit TestStand and Migrating from 32-bit TestStand](#)
- [Process Model Plug-In Architecture](#)
- [TestStand Offline Results Processing Utility](#)

Related tasks:

- [Manually Migrating Result Processing Options from Any Version of TestStand](#)

Related information:

- [Legacy Model Switcher](#)
- [Results Processing Dialog Box](#)
- [TestStand Migration Utility Environment Reference](#)

Process Model Changes in TestStand 2012 or Later and the Migration Utility

The TestStand 2012 or later process models include significant changes to implement a plug-in architecture for model sequence files. TestStand 2012 or later uses the plug-in mechanism to decouple result processing from the process models. As part of this change, the result processing options in TestStand 2010 SP1 or earlier move from `TestStandModelReportOptions.ini` and `TestStandDatabaseOptions.ini` into a single `ResultProcessing.cfg` file, located in the `<TestStand Application Data>\Cfg\ModelPlugins` directory, that stores configuration information for all result processing model plug-ins. The result processing options files from TestStand 2010 SP1 or earlier are not compatible with TestStand 2012 or later. Use the TestStand Migration Utility to migrate the result processing options from TestStand 2010 SP1 or earlier installed on your

computer to the current version of TestStand.



Note The utility does not support migrating configuration files from a previous version of TestStand if you modified the **ReportOptions** or **DatabaseOptions** data types defined in the default process models.

Use the following migration actions in the utility to migrate the result processing configuration options from TestStand 2010 SP1 or earlier installed on the computer to the current version of TestStand:

- **Apply to new models**—Creates a new configuration in a configuration set named **Migrated from TestStand Version**. The utility automatically selects a unique name for the configuration if the utility has already used the default name in a previously completed migration. Refer to the migration report for the specific configuration names the utility uses.
- **Apply to legacy models**—Copies the options files from TestStand 2010 SP1 or earlier to the current TestStand directory so you can use the previous configurations with the equivalent legacy TestStand 2010 process models in TestStand 2012 or later. When you migrate database options, the utility renames `TestStandDatabaseOptions.ini` to `TestStandDatabaseSchemas.ini`.
- **Apply to new and legacy models**—Adds a new configuration in the **Migrated from TestStand Version** configuration set and copies the options files from TestStand 2010 SP1 or earlier.

If you encounter an incompatibility in an existing test system when you use the current TestStand process models or other circumstances prevent you from using the current TestStand process models, you can use the equivalent legacy TestStand 2010 process models. However, National Instruments recommends using the current TestStand process models in most cases.



Note If you migrate a custom legacy process model file to TestStand 2012 or later, you must also migrate the corresponding `ModelSupport.seq` file from TestStand 2010 SP1 or earlier, or you must use the equivalent legacy TestStand 2010 process models, which include a legacy `ModelSupport.seq` file.

Related concepts:

- [Search Directories](#)

Related tasks:

- [Manually Migrating Result Processing Options from Any Version of TestStand](#)

Nigel, AI Advisor

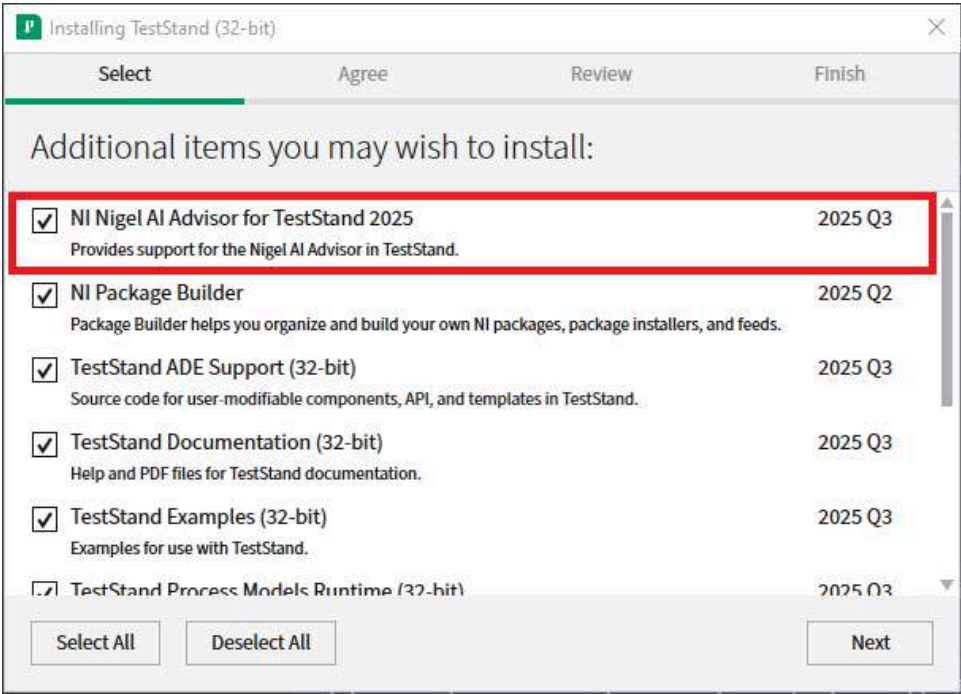
Nigel is a design assistant powered by Microsoft Azure OpenAI. Nigel brings a wealth of knowledge about TestStand into your design process to help you create projects and sequences.

Nigel can help you while you are working in TestStand by performing tasks like:

- Finding examples to give you a solid starting point for a project
- Providing instructions for completing a specific task
- Explaining key TestStand and sequence design concepts
- Assisting with finding specific settings or dialog boxes in the TestStand environment
- Summarizing documentation topics, and
- Answering questions about hardware functionality and configuration.

Installing Nigel

Nigel is installed by default while installing TestStand. If you do not want to install Nigel, you can deselect **NI Nigel AI Advisor for TestStand <Version Year>** when setting up your TestStand installation.



Accessing Nigel

Nigel is integrated into the TestStand Sequence Editor, making it easy to open a chat window with Nigel from the toolbar.

Figure 2. Nigel Access Location

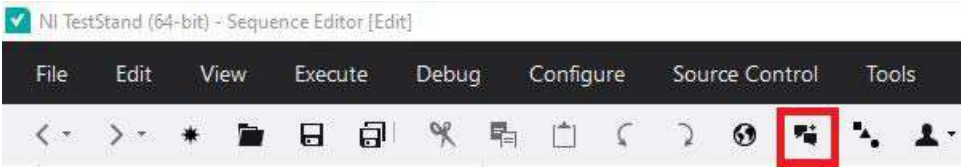
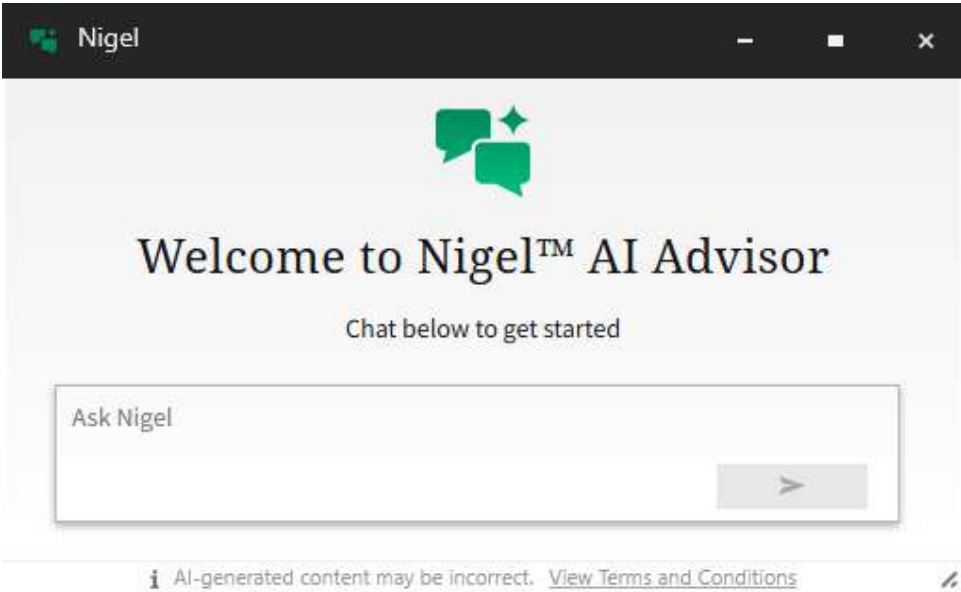


Figure 3. Nigel Chat Window



After you open Nigel, you must log in to your `ni.com` user account and ensure it is associated with an active subscription license, multi-seat license, or Software Service Program for TestStand Development System.

Rate Limits

Access to Nigel is rate limited per user and is measured as Requests Per Day (RPD). The RPD limit is shared between the TestStand chat and all LabVIEW Nigel search functions.

Disabling Nigel

You can disable Nigel features by editing the following registry keys:

- You can disable the Discuss with Nigel feature by setting the `Computer\HKEY_LOCAL_MACHINE\SOFTWARE\National Instruments\Nigel AI Advisor\DisableNigelSharing` registry key (DWORD (32-bit), 0 or 1) to 1.
- You can disable all Nigel features, including Discuss with Nigel, chat, and search, by setting the `Computer\HKEY_LOCAL_MACHINE\SOFTWARE\National Instruments\Nigel AI Advisor\DisableNigel` registry key (DWORD (32-bit), 0 or 1) to 1.

Related information:

- [Meet Nigel, NI's New AI Advisor](#)

Deploying TestStand Systems

This section contains information about the following concepts:

- Evaluating deployment-related concepts before you design a test system.
- Selecting the most effective mechanism for creating one or multiple deployments depending on the functional and maintenance requirements of the test system and related deployment.
- Using the TestStand Deployment Utility to create a customized full test system deployment or patch deployment.

The NI TestStand Advanced Architecture Series is a suite of documents, located on the NI website, authored by TestStand architects and developers to provide more advanced concept and architecture information for experienced TestStand users with complex projects.

Related concepts:

- [Before You Begin](#)
- [Deployment Process Overview](#)
- [Choosing a Deployment Mechanism](#)
- [Choosing Single or Multiple Deployments](#)
- [Customizing Components You Deploy](#)
- [Patching Deployments](#)
- [TestStand Directory Structure](#)

Before You Begin

Before you start developing a test system, you must understand the concepts in this section so you can make appropriate design decisions that affect deployment as you complete development tasks.

TestStand systems typically run on test station computers used exclusively for testing devices during the design validation or manufacturing process. Test system developers use separate development computers to create and modify code modules, test sequences, user interfaces, and process models without affecting tests that run on test

station computers. In most cases, development computers contain the complete test system, including NI drivers and components and any required third-party components and installers.

Typically, developers manually configure test systems on development computers by running multiple installers, copying files, and changing settings as required. However, using a manual technique might not be efficient or reliable when you must move a test system from a development computer to multiple test station computers.

Deployment

Deployment is the process of moving software components a TestStand-based test system requires for successful execution from a development computer to one or more test station computers. Using a well-designed and robust deployment process, you can more easily accomplish the following tasks:

- Increase the number of test station computers, such as adding test station computers to a production line
- Update deployed test systems to correct issues or add support to test new products or product lines
- Develop new test systems that reuse existing components

NI provides two applications you can use to deploy TestStand systems:

- **TestStand Deployment Utility** – Allows you to create MSI-based distributions and patches, create simple package-based distributions and patch installers, and create deployable images shared via network drive or source code control system.
- **NI Package Builder** – Allows you to create multiple top-level packages from a single TestStand project and build multiple distributions.



Note

- You may need to update NI Package Builder to support building packages for the version of TestStand you use. Refer to the TestStand Readme and the NI Package Builder Readme for information about supported versions.
- NI Package Builder is missing some features that the TestStand Deployment Utility supports. NI plans to continuously develop new

features in Package Builder so that it can eventually replace the TestStand Deployment Utility. New releases of Package Builder will be made available through NI Package Manager.

- If your deployment applications require any missing features of NI Package Builder, you can continue to use the TestStand Deployment Utility until the features are added to Package Builder. Refer to the ***NI Package Builder Manual*** on ni.com/docs for a comparison of features in TestStand Deployment Utility versus Package Builder.

Use the appropriate application to complete the following common tasks:

- Specify source files directory or a TestStand workspace file that references required files and dependencies, including sequence files, code modules, and other supporting files
- Analyze test sequences and create a full or patch deployable image of the test system
- Use the deployment utility or third-party installer development tools to build a customized full or patch deployment distribution to transfer the test system from the development computer to the test station computer
- Use the deployable image as the source for deploying test systems using a network drive or a source code control system

Using MSI-Based Installers to Deploy Test Systems

For MSI-based deployments, you must use the TestStand Deployment Utility. You can use the deployment utility to accomplish the following installer-related tasks:

- Build a single, custom installer to redistribute only the NI drivers and software components that the test system requires and that you have installed on the development computer.



Note You must install the TestStand Runtime for TestStand-based test systems to function correctly. You must also install the LabVIEW Run-Time Engine on the test station computer to execute VIs and any required NI hardware devices drivers, such as NI-DAQmx.

- Build multiple installers for the test system to reduce the time required to create the deployment, to reduce the size of the installer, and to more easily perform frequent updates for the test system you deploy.
- Install multiple test systems on a single test station computer.
- Build patch deployment installers that contain only the files and software components that differ from the previous deployment.



Note Refer to <National Instruments>_Legal Information for information about including legal information in installers built with NI products.

Using Package-Based Distributions to Deploy Test Systems

You can use a package based distribution to distribute as a package visible from NI Package Management tools, including NI Package Manager. You can use the package distribution capability of the TestStand Deployment Utility or NI Package Builder to accomplish the following tasks:

- Build a single package file, which you can distribute by copying directly to a test station computer, or using NI package sharing tools.



Note You must install the TestStand Runtime for TestStand-based test systems to function correctly. You must also install the LabVIEW Run-Time Engine on the test station computer to execute VIs and any required NI hardware devices drivers, such as NI-DAQmx.

- Create a repository or package installer, which includes required run-time and driver dependencies in the distribution in addition to the single package file.
- Build a patch deployment which includes only updated packages for drivers and run-time software that were not present in the previous deployment.



Note To create multiple top-level packages from a single TestStand project, NI recommends using a supported version of NI Package Builder. Refer to ***Building a Distribution with NI Package Builder*** and the ***NI Package Builder Manual*** on ni.com/docs for more information.

Related concepts:

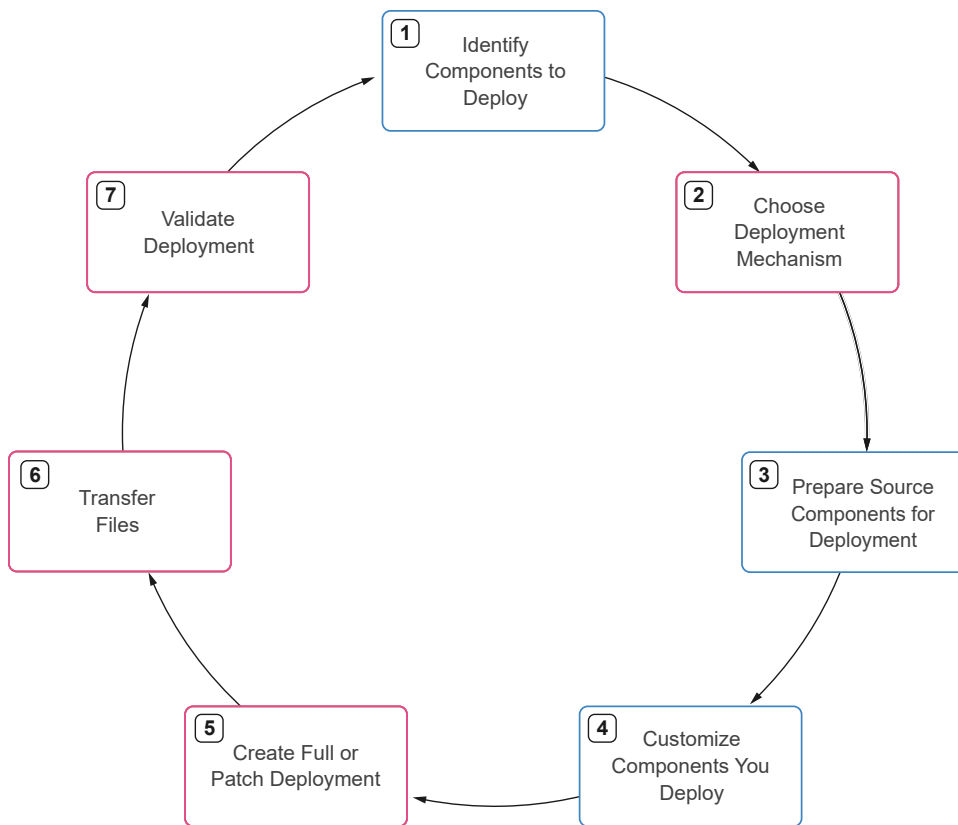
- [Deployment Process Overview](#)
- [Preparing Source Components for Deployment](#)
- [Using a Directory to Create a Deployment](#)
- [Using a TestStand Workspace File to Create a Deployment](#)
- [Identifying Components to Deploy](#)
- [Analyzing the Test System](#)
- [Creating Deployments](#)
- [Building an Installer with the TestStand Deployment Utility](#)
- [Building an Installer with Third-Party Tools](#)
- [Building a Customized MSI-based Installer](#)
- [Transferring Files](#)
- [Using a Network Drive to Deploy Test Systems](#)
- [Using a Source Code Control System to Deploy Test Systems](#)
- [Redistributing MSI-based Installer Products in a Deployment](#)
- [Choosing Single or Multiple Deployments](#)
- [Deploying Multiple Test Systems to the Same Test Station](#)
- [Patching Deployments](#)
- [Building a Package Distribution with NI Package Builder](#)
- [Deploying 32-bit TestStand and 64-bit TestStand Systems](#)
- [Installer Issues for 32-bit TestStand and 64-bit TestStand](#)

Related information:

- [NI Package Builder Manual](#)

Deployment Process Overview

Complete the following tasks to carefully plan, design, implement, execute, and validate a full or patch deployment process for a complex TestStand-based test system.

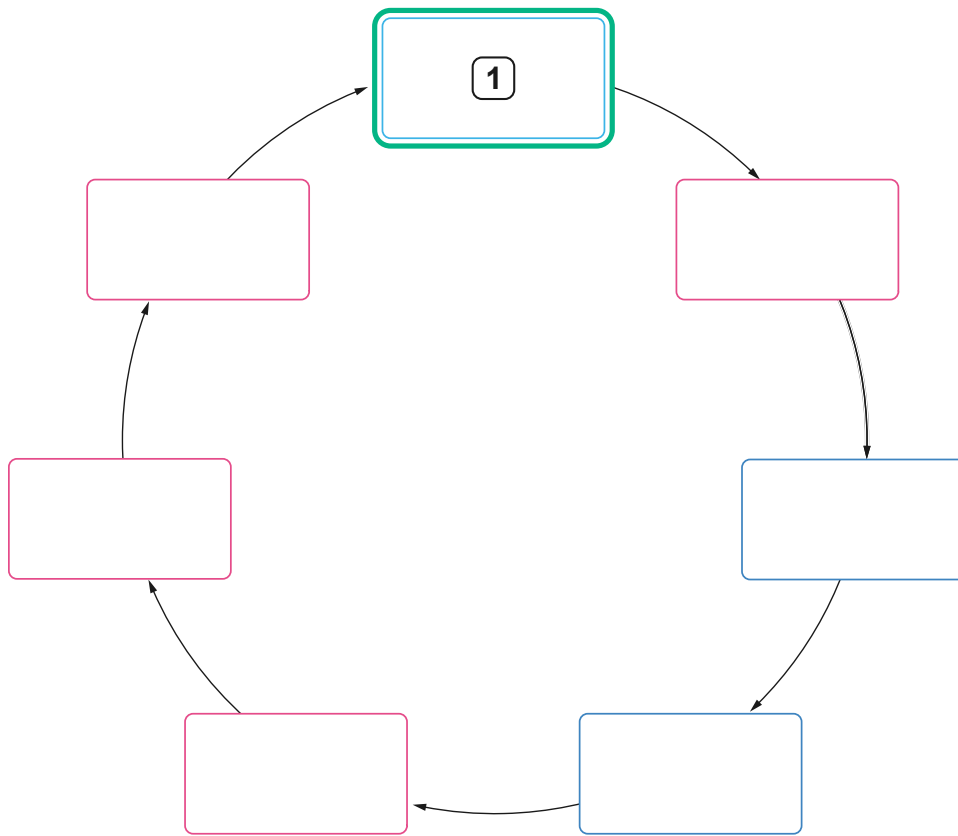
**Legend**

1. [Identifying Components to Deploy](#)
2. [Choosing a Deployment Mechanism](#)
3. [Preparing Source Components for Deployment](#)
4. [Customizing Components You Deploy](#)
5. [Creating Deployments](#)
6. [Transferring Files](#)
7. [Validating Deployments](#)

Related concepts:

- [Patching Deployments](#)

Identifying Components to Deploy



Use the following table to identify the files to explicitly include in the deployment to ensure that the test system executes successfully on the test station computer:

Component	Guidance
TestStand license	Review the TestStand license options to determine which license to use.
Sequence files	Identify the set of sequence files the test system requires by creating a list of the high-level tasks the test system performs, including trouble-shooting, calibrating devices, and performing other diagnostic tasks. For each task, identify the root sequence or sequences that accomplish the task. For each sequence, identify the process models the sequence uses. The TestStand Deployment Utility automatically identifies and includes sequence files for the sequences that Sequence Call steps statically reference. You must manually identify and include sequence files for sequences that Sequence Call steps dynamically reference. Use the Sequence Hierarchy window in the TestStand Sequence Editor to display a graph of the hierarchy of all sequences and sequence calls that use expressions for a sequence file.

Component	Guidance
TestStand Runtime	A test system you deploy to a test station computer requires the TestStand Runtime to execute TestStand sequence files. You can customize the features of the TestStand Runtime to include in the installer you build with the deployment utility, such as excluding the default TestStand examples and tutorials. The TestStand Engine is part of the TestStand Runtime. The TestStand Engine includes a backward-compatible API so you can use existing property and method calls in later versions of the TestStand Engine without modification and so you can run user interfaces built in earlier versions of TestStand with later versions of the TestStand Engine.
Code modules	The deployment utility automatically includes static dependencies of code modules, such as utility DLLs statically linked to code module DLLs and VI dependencies. However, you must manually include implicitly referenced files, such as documentation, images, and support files, and dynamically referenced files, such as VIs you call using VI Server or dynamically loaded DLLs. You must verify that the test system uses all the dynamically called files you added to the workspace or to destination directories, including code modules you added to a workspace by using the Insert Code Modules context menu item in the Workspace Browser dialog box or the Insert Code Modules context menu item in the Workspace pane.
Files that Property Loader steps call	The deployment utility automatically includes source files that Property Loader steps reference. If the file is specified dynamically through a TestStand expression that can only be evaluated at runtime, you must manually include all the files the expression could potentially reference. TestStand Deployment Utility will display a message listing any expressions that it could not evaluate.
Image files, support files, configuration files, or data files	Because other files in a test system typically implicitly reference image, support, configuration, and data files, you must manually include these types of files.
LabVIEW toolkits	Most LabVIEW toolkits do not require the toolkit license to execute VIs you build with the toolkit. Refer to the toolkit licensing requirements to determine if you need to include the toolkit license or any additional components required to execute these VIs.
LabWindows/CVI Run-Time Engine	You can use the shared or side-by-side version of the LabWindows/CVI Run-Time Engine (RTE) to execute DLLs built with LabWindows/CVI from TestStand.
LabVIEW Run-Time Engine	By default, the deployment utility includes all required dependencies and compiles all VIs included in the deployment. You must manually include any

Component	Guidance
	dynamically-called VIs in the deployment.
LabVIEW NXG Run-Time Engine	By default, the deployment utility invokes LabVIEW NXG to recompile all GLL binaries included in the deployment using the associated source files. You can optionally include source files in the deployment to make changes to LabVIEW NXG code modules on the test station computer. The LabVIEW NXG run-time engine is required to execute test code within GLLs.
LabVIEW modules	The LabVIEW DataLogging and Supervisory Control (DSC) Module requires the LabVIEW DSC Module Run-Time System installed on the test station computer. In contrast, you can run LabVIEW Real-Time Module VIs on a pre-configured, real-time computer without including the LabVIEW Real-Time Module in the deployment. Refer to the documentation for the LabVIEW module to determine whether you need to include it.
Configuration files in the <TestStand Application Data>\Cfg directory	Include the configuration files you customized for the settings you want to use on the test station computer. You can use configuration files customized for test sequence developers to use on development computers that are different from configuration files you customize for operators to use on test station computers.
TestStand components	Include the components you customized in the <TestStand Public>\Components directory, such as a process model you modified to launch a customized dialog box for entering scanned UUT serial numbers or a user interface you customized to provide debugging information for operators. Enable the From TestStand Public Directories option in the Deploy Files section on the System Source tab of the deployment utility to include all files in the <TestStand Public>\Components directory. If you deploy a process model based on an NI model, the model uses a language resource file to support localized strings. For model strings to appear correctly, you must also include the required language file in your deployment:<TestStand>\Components\Language\English\ModelStrings.ini. To ensure this file is included, copy it to the <TestStand Public>\Components\Language\English directory before deploying.
User interface executable	You can deploy the default user interfaces TestStand includes or you can create a customized user interface to deploy. If you deploy a customized user interface, include the following files:

Component	Guidance
	• The user interface executable • Configuration files • Dependent DLLs, ActiveX controls, or .NET assemblies or controls • Language .ini files for localization, if required • Required run-time engines
Run-time engines	The following file types require certain components on the test station computer to execute correctly. You do not need to include the required component in the deployment if the test station computer already has the component installed. • Code modules, executables or DLLs you built with LabVIEW— ◦ Version of LabVIEW RTE that matches the version of LabVIEW you used to create the code module, executable, or DLL ◦ You can install multiple versions of the LabVIEW RTE on a test station computer. However, NI recommends using only one version of the LabVIEW RTE in a single test system to simplify deployment. • Code modules, executables or DLLs you built with LabVIEW NXG— ◦ Version of LabVIEW NXG RTE that matches the active version of LabVIEW NXG you used to rebuild the GLL binaries ◦ You can install multiple versions of the LabVIEW NXG RTE on a test station computer. However, NI recommends using only one version of the LabVIEW NXG RTE in a single test system to simplify deployment. • DLL code modules or executables you built with LabWindows/CVI— ◦ Version of LabWindows/CVI RTE that matches or that is later than the version of LabWindows/CVI you used to create the executable or DLL ◦ ▪ You can install or upgrade only one version of the default LabWindows/CVI RTE on a test station computer at a time, but you can install or upgrade multiple versions of the side-by-side version of the LabWindows/CVI RTE. ▪ When you deploy a user interface executable or a DLL module that is bound to a side-by-side version of the LabWindows/CVI RTE, you must explicitly include the side-by-side version of the LabWindows/CVI RTE in the deployable image you use to build an installer with the deployment utility. In the Drivers and Components dialog box, you must enable the item in the Components to Include list that corresponds to the side-by-side version of the LabWindows/CVI RTE you want to use, such as NI LabWindows/CVI Side-By-Side Run-Time Engine 2012. • .NET assembly—Version of .NET that supports executing the assembly (Refer

Component	Guidance
	to Microsoft documentation for more information about .NET version compatibility and deployment.)
NI hardware drivers	Refer to the documentation for the NI device the test system uses to determine the driver to include. You do not need to include the NI hardware driver in the deployment if the test station computer already has the driver installed. You can also include NI hardware configuration information.
Third-party components	Include the third-party hardware drivers or software components required to execute the test system correctly. You can include the collection of third-party files in the single installer you build with the deployment utility using custom commands.

Components to Deploy for Editing on a Test Station

In some cases you might need to make changes to debug or fix an issue with the test system. Typically, after you correct the issue with the test system, you re-deploy the entire test system or you deploy a patch that contains only the updated module.

If you want to edit code modules on the test station computer, you must install the LabWindows/CVI Development System to debug or recompile DLLs you built with LabWindows/CVI, and you must install the LabVIEW Development System and all toolkits and modules required to debug or edit VIs. You can use a TestStand Debug Deployment Environment License on the test station computer to edit code modules on the test station computer.

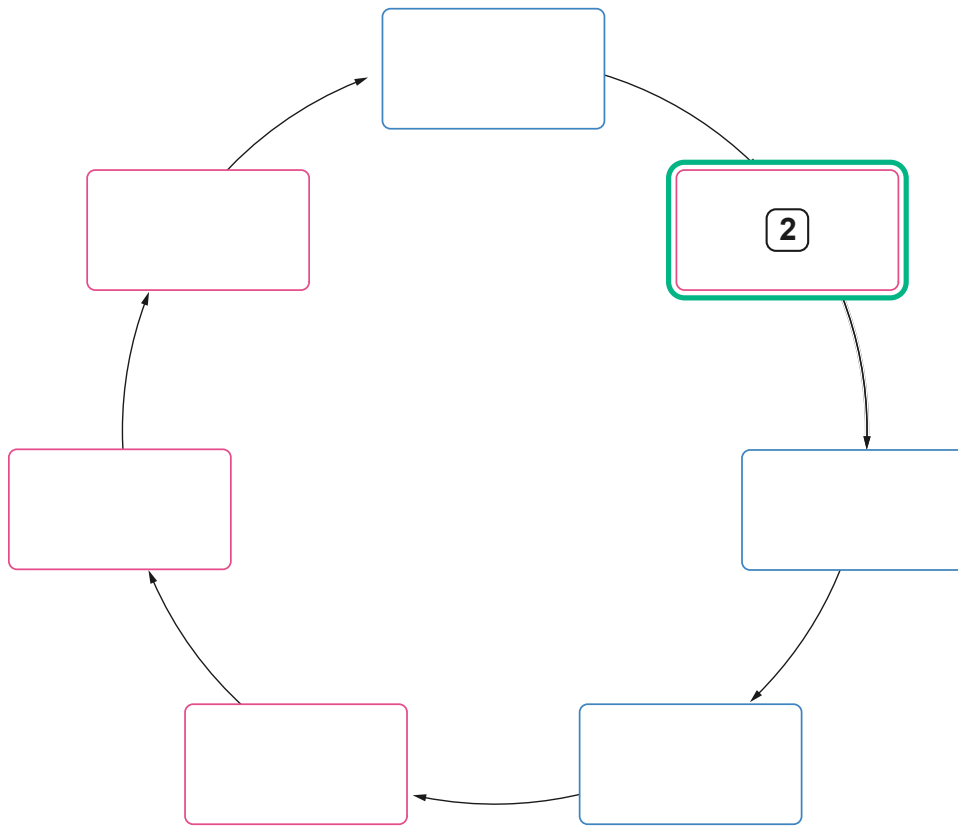
Exclude VIs from `vi.lib`, `instr.lib`, or `user.lib` from deployments when you deploy VIs that you expect others to edit, such as when you deploy VIs to another development computer or to a test station computer that includes the LabVIEW Development System, the TestStand Runtime, a custom user interface, and a TestStand Custom Sequence Editor License so other developers can modify and redeploy test sequences.

To include source files for LabVIEW NXG code modules, enable the **Include LabVIEW NXG source files** option, located in the **LabVIEW NXG options** tab of the LabVIEW VI options dialog. Launch the LabVIEW VI options dialog using the LabVIEW Options button on the Distributed Files tab.

Related concepts:

- [Including Source Components](#)
- [Activating and Licensing TestStand](#)
- [Manually Adding or Removing Files to or from Deployments](#)
- [Choosing Between the Shared or Side-by-Side Version of the LabWindows/CVI Run-Time Engine](#)
- [Including TestStand Configuration Files in a Deployment](#)
- [TestStand Directory Structure](#)
- [Configuration Files](#)
- [Including Files from the TestStand Public Directory in a Deployment](#)
- [User Components](#)
- [Including a User Interface in a Deployment](#)
- [Creating Custom User Interfaces](#)
- [Using Side-by-Side Versions of the LabWindows/CVI Run-Time Engine with TestStand](#)
- [Building a Customized MSI-based Installer](#)
- [Including Hardware Configuration Information in a Deployment](#)
- [Using Custom Commands to Execute Third-Party Installers](#)
- [Customizing Components You Deploy](#)

Choosing a Deployment Mechanism



You can use the following installer-based or network-based mechanisms to transfer a test system to one or more test station computers:

- Distribution you build with the TestStand Deployment Utility
- Distribution you build with NI Package Builder
- Installer you build with third-party tools
- Network drives
- Source code control (SCC) system/revision control system (RCS)

Consider the following issues as you determine the to use:

- **Network access of test station computers**—If no network infrastructure exists, you cannot use a network drive or SCC system to deploy test systems and must instead use an installer on distribution media. Using the deployment utility instead of a third-party tool to build a distribution requires less specialized knowledge and development time and offers flexible customization options, such as using custom commands to run other installers. You must use a third-party tool to build

installers that support repairing an installation or allowing users to select the features to install.

- **Test system functionality during network outage**—If test station computers must maintain production-level operation during network outages, the test station computers must include local copies of all the files the test system requires, such as user interfaces, configuration files, sequence files, code modules, files shared among test station computers, and so on. Test station computers must also locally store reporting and database logging results if you must maintain test system functionality during a network outage. You can deploy the test system using a network drive or an SCC system, but you must store local copies of the files on the test station computer.
- **Initial development investment and updating requirements**—Use the deployment utility to build a distribution to deploy a test system to test station computers you do not plan to update frequently. You can use the deployment utility to build patch distributions to update these types of test stations. Use a network-based deployment mechanism to deploy a test system to a large number of test station computers you plan to update frequently. Setting up a network-based solution for a large deployment can make updating the test system more efficient. You can also use a combination of mechanisms to deploy a test system. For example, you can use the deployment utility to build distributions and transfer those distributions to test stations using a network drive or SCC system.
- **Design the test system to account for the deployment mechanism**—You can automate the install and update process by writing an application that downloads and uses all the necessary distributions to set up the test system instead of running several deployments manually for each test station. You can also run a batch program that synchronizes to a set of directories from an SCC server or copies a set of directories from a network drive and then runs the downloaded distributions. If the test system requires a manual initial set-up process, you can automate updating the test code. For example, you can modify a process model to read the UUT serial number and synchronize sequence files or code modules from an SCC server or copy the files from a network drive before testing the UUT. Consider the following issues if you use the Microsoft Windows startup folder or Task Scheduler to launch an update application you create to complete updates before or after testing occurs.
 - If you implement an automatic update solution, design the application to fail without changing the test system or revert the changes to account for situations in which network connections fail. Network outages during an

- update can result in a partially updated test system, which might return incorrect results or not work properly.
- Copy all the updated installers to the test station before running the installers to prevent partial updates. If any copy of an installer fails, do not run any of the installers.
- Revert changes by backing up directories before syncing the directories to the SCC server. If any synchronization fails, restore the directories from the backup copies.
- If the test system requires a manual update process, you can use the user interface to prompt users to perform an update after a certain amount of time passes.
- **Creating patch deployments**—Use the same deployment mechanism for patch deployments as you used for the full deployment you want to update.

If you choose to create a distribution with the TestStand Deployment Utility, you must consider the following build output options:

- **Deployable Image Only**—A directory which contains the files and statically referenced dependencies you specify on the System Source tab.
- **MSI-based Installer**—A custom installer to redistribute the deployable image and any NI drivers and software components required by the deployment.
- **Package-based distribution**—A custom NI package to redistribute the deployable image. Package based distributions can reference or include required driver and run-time packages depending on the distribution type.



Note To create multiple top-level packages from a single TestStand project, NI recommends using a supported version of NI Package Builder. Refer to ***Building a Distribution with NI Package Builder*** and the ***NI Package Builder Manual*** for more information.

If you do not use the deployment utility to build a distribution, you can use the deployment utility to create a deployable image, which you can deploy to test station computers by using a different deployment mechanism.

Regardless of the deployment mechanism you use, NI recommends that you document the required process and procedures to effectively create a deployment.

The following table lists the important features each deployment mechanism supports.

Feature of Deployment Mechanism	Distribution You Build with Deployment Utility or NI Package Builder	Installer You Build with Third-Party Tools	Network Drive	SCC System
Provides a simple way to create a deployment that requires only configuration, not additional development tasks.	✓	—	—	—
Can deploy the TestStand Runtime, which is required to execute TestStand-based test systems. Mechanisms that cannot automatically deploy the TestStand Runtime must install TestStand by running the installer NI provides.	✓	—	—	—
Can deploy NI drivers and components, such as NI-DAQmx. Mechanisms that cannot automatically deploy NI drivers must install the drivers by running the installers NI provides, such as the NI Driver DVD.	✓	—	—	—
Can import hardware configuration information into Measurement & Automation Explorer. You must manually import hardware configuration information if the mechanism does not support automatically importing the information. Using an installer you build with third-party tools requires additional development to import the configuration.	✓	✓	—	—
Can run multiple third-party installers. You must manually run third-party installers if the mechanism does not support automatically running the installers.	✓	✓	—	—
Can install files in TestStand directories, such as the <TestStand Application Data>\Cfg directory, the <TestStand Public> directory, and the <TestStand> directory.	✓	—	—	✓
Can create a deployment without having specific knowledge of Microsoft Windows Installer technology.	✓	—	✓	✓

Feature of Deployment Mechanism	Distribution You Build with Deployment Utility or NI Package Builder	Installer You Build with Third- Party Tools	Network Drive	SCC System
Does not require test station computers to rely on a network infrastructure.	✓	✓	—	—
Test station computers use local copies of files to ensure test system operation during network outages.	✓	✓	—	✓
Submitting modified files does not affect executing test systems because test station computers receive changes only when requested.	✓	✓	—	✓
Supports uninstalling files you installed for the test system.	✓	✓	—	—
Supports repairing the deployment to replace deleted or modified files.	—	✓	—	—
Provides a way for users to decide which features to install on a test station computer.	—	✓	—	—
Provides an efficient way to perform small updates to the deployment. The size of the files changed determines the size of the update, which does not require a large, fixed, overhead size.	✓	✓	✓	✓
Automatically identify the set of modified files to include in a patch deployment.	✓*	—	—	✓
Easily scales to a large number of test system computers by updating files in a central repository.	—	—	✓	✓
Can revert to a previous version of the test system.	—	—	—	✓

Features marked with an asterisk (*) are available in MSI-based installer deployments created by TestStand Deployment Utility, but are not available in package-based deployments.

Related concepts:

- [Network-Based Deployment Mechanisms](#)
- [Building an Installer with the TestStand Deployment Utility](#)
- [Building a Package Distribution with NI Package Builder](#)
- [Building an Installer with Third-Party Tools](#)
- [Using a Network Drive to Deploy Test Systems](#)
- [Using a Source Code Control System to Deploy Test Systems](#)
- [Building a Customized MSI-based Installer](#)
- [Using Custom Commands to Execute Third-Party Installers](#)
- [Creating a Deployable Image](#)
- [TestStand Directory Structure](#)
- [Relationship of NI MSI-based Installer Architecture and Microsoft Windows Installer Technology](#)
- [Uninstalling a TestStand Deployment](#)

Related information:

- [NI Package Builder Manual](#)

Choosing a TestStand Deployment Utility Deployment Build Output

Familiarize yourself with the different output types before using this help topic:

- Creating a deployable image
- Building an Installer with the TestStand Deployment Utility
- Building a package distribution with the TestStand deployment utility

Feature Comparison for TestStand Deployment Utility build output types

Feature	Deployable Image Only	MSI Based Installer	Package Based Distribution - Single Package	Package Based Distribution - Repository	Package Based Distribution - Package Installer
Consolidates all files in the deployment into a single file	-	-	Yes	Yes	Yes
Can be installed by running a single	-	Yes	Yes	No	Yes

Feature	Deployable Image Only	MSI Based Installer	Package Based Distribution - Single Package	Package Based Distribution - Repository	Package Based Distribution - Package Installer
application					
Deploy files to multiple destinations on a test station computer	-	Yes	Yes*	Yes*	Yes*
Allow the user to specify a custom target directory for files while installing the deployment on a test station computer	Yes	Yes	-	-	-
Allows registration of executables or DLL as activeX servers on the test station computer	-	Yes	-	-	-
Allows creation of program items in the start menu for files in the deployment	-	Yes	-	-	-
Allows creation of shortcuts to files in the deployment	-	Yes	Yes, with restrictions ¹	Yes, with restrictions ¹	Yes, with restrictions ¹
Automatically import hardware settings using a MAX configuration	-	Yes	-	Yes, with restrictions ²	Yes, with restrictions ²
Include custom commands to run 3rd party tools or executables	-	Yes	-	-	-
Create a file association for TestStand sequence	-	Yes	-	-	-

Feature	Deployable Image Only	MSI Based Installer	Package Based Distribution - Single Package	Package Based Distribution - Repository	Package Based Distribution - Package Installer
files to a specific User Interface application					
Available in 32-bit and 64-bit TestStand	Yes	Yes	Yes	Yes	Yes
Supports installation with no user interaction (silent installation)	-	Yes	Yes	Yes	Yes
Can be installed on a clean machine without NI software	Yes	Yes	No, NI Package Management software and required drivers must be present	No, NI Package Management software must be present	Yes
Includes required drivers and run-times in the distribution	-	Yes	No, dependencies are referenced only	Yes	Yes
Require a specific version or version range for required drivers and components	-	No, but included drivers will be installed if an older version is present on the deployment target	Yes	Yes	Yes
Include references to recommended or optional drivers or components	-	-	Yes	Yes	Yes
Create patch distributions which contain only new or	-	Yes	-	-	-

Feature	Deployable Image Only	MSI Based Installer	Package Based Distribution - Single Package	Package Based Distribution - Repository	Package Based Distribution - Package Installer
updated files					
Create patch distributions which contain only new or updated drivers and components	-	Yes	Yes	Yes	Yes
Include MSI-based additional components	-	Yes	-	-	-
Include package-based additional components	-	Yes, but only automatically detected dependencies	Yes	Yes	Yes
Distribute deployment with NI package distribution tools	-	-	Yes	Yes	Yes

¹Shortcuts in package deployments do not support the **command line arguments** and **start in** options

²MAX configuration imports from package based distributions do not support silent installation

Choosing the Best Build Output for Your Deployment Scenario

If multiple deployment output types are viable based on the table above, consider the below recommendations to choose the best option for your requirements.

1. The deployment requires specific software versions of drivers and components to meet validation requirements. **Recommendation:** Create a package based deployment and configure the package dependencies to require a specific version of all drivers. Create a package installer or repository to include the required

versions of the dependencies in the software.

2. The deployment will be installed on many test station computers with network access, and each must have the same version of driver and components.

Recommendation: Use the repository build output to generate a feed which references all required drivers. Host the feed and packages in a network location where it is accessible by the test station computers. Use package management software to register the feed on the test station computer to access the packages.

3. The deployment will be installed on many test station computers without network access, and each must have the same version of driver and components.**Recommendation:** Create an MSI-based installer with all required software, and ensure that test station computers do not have any NI software present before installing the deployment.

Deploying with Environments Using the TestStand Deployment Utility

Deploying with Environments

The TestStand Deployment Utility supports using an engine environment file (`.tsenv`) to deploy files from a custom <TestStand Public> directory or <TestStand Application Data> directory. In addition, the TestStand Deployment Utility allows you to specify an environment file for use on the deployment machine. If you do, files you deploy to the <TestStand Public> directory or <TestStand Application Data> directory will be deployed to the custom locations specified in the environment file. For more information about creating and using environment files, refer to the TestStand Environments topic.

Using Custom Environment Directories on the Development Machine

To use a custom TestStand Public directory and/or Application Data directory on the development machine, specify an environment file in the **Engine Environment** field of the System Source tab. Once you have specified an environment file, select the **From TestStand Public Directories** or **From TestStand Configuration Directory** to deploy files from the directory specified in the environment file. If you create a deployment with these settings, files will be deployed to the default TestStand Public and Application Data directories.



Note The `Users.ini` file, which contains TestStand user management

settings, always exists in the global location, even if an environment file specifies a different path for the Application Data directory.

Related concepts:

- [TestStand Environments](#)

Related information:

- [TestStand Environment Reference Help](#)

Deploying to Custom Environment Directories

To deploy files to your custom TestStand Public directory and/or Application Data directory on the deployment machine, select the **Use Environment File To Determine Deployed File Destinations** option. After selecting this option, use the Destination View in the Distributed Files tab to preview the destinations for files deployed to the TestStand Public or Application Data destinations. The custom destination path is displayed in parentheses after the destination name.

The TestStand Deployment Utility does not support using absolute paths in the environment file when used to determine file destinations. Paths you specify must be relative to the location of the environment file on the destination machine. The location of the custom directories on the target machine is represented by the following combination:

```
<destination for environment file> + <custom path specified in environment file>
```

To access the custom directories on the deployment machine, your deployed User Interface must use the deployed environment file. You can use the command line flag `/env <Environment File Path>` when launching a user interface executable to configure the environment file it uses upon launch. Follow these steps specify that the deployment utility create a shortcut to a user interface with this flag:

1. In the Distributed Files tab, select the user interface executable.
2. Click the **Shortcuts** button to launch the shortcuts dialog box.
3. Click **Add Shortcut** to create a new shortcut.
4. Set the shortcut base path to the same base path you selected for the environment

file.

5. Set the shortcut relative path to the same relative path of the environment file.
6. Set the command line arguments to `/env <EnvironmentFilename>`, where `<EnvironmentFilename>` is the name of the environment file you selected on the System Source tab.
7. Ensure that the user interface is deployed to the same destination as the environment file.

Using a Different Environment File on the Deployment Machine

You can configure the TestStand Deployment Utility to use a different environment file to determine the custom locations for the TestStand Public directory and/or Application Data directory.

Follow these steps to configure an environment file for use on the deployment machine:

1. In the Distributed Files tab, select the environment file you specified in the **Active Engine Environment** field.
2. Use the browse button for the **Replacement Path** field to select a different environment file.

Once the file is replaced, the custom destination paths will update to reflect the paths in the selected file.

Building an Installer with the TestStand Deployment Utility

You can use the TestStand Deployment Utility to build an installer that contains the files the test system requires to execute successfully on a test station computer, such as sequence files, support files, user interface files, the TestStand Runtime, NI run-time engines (RTEs), and NI drivers and components. Additionally, the installer you build with the deployment utility can perform the following tasks:

- Import hardware configuration information from Measurement & Automation Explorer (MAX)
- Register ActiveX Automation servers
- Create program items, such as shortcuts
- Launch a Readme file and license agreement at install time

- Associate sequence files (`.seq`) with a user interface by using the Alternate Association for `.seq` Files option in the Advanced Installer Options dialog box
- Run other installers

The deployment utility uses options and configuration settings to build an installer based on Microsoft Windows Installer technology but does not require you to have specific knowledge of Windows Installer technology.



Note Refer to `<National Instruments>\_Legal Information` for information about including legal information in installers built with NI products.

Advantages

Using an installer you build with the deployment utility offers the following advantages:

- Using the deployment utility to build the installer does not require additional development tasks to build or customize an installer. You configure installer options in the deployment utility to build an installer using Windows Installer technology.
- You can customize the installer to include the TestStand Runtime, RTEs, and NI drivers and components. Because you select the TestStand Runtime optional features to include at build time, you can ensure that you deploy the same set of TestStand Runtime optional features to all test station computers.
- The installer can launch third-party installers.
- You can include MAX hardware configuration information from the development computer to import into MAX on the test station computer.
- You can place the installer on removable distribution media to physically move to test station computers, which do not require a network infrastructure.
- You can install files in TestStand-specific directories, such as the `<TestStand>` or `<TestStand Public>` directories, even if you do not install TestStand to the default location.
- The installer can uninstall files it installed on the test station computer.
- You can build a patch installer that installs only an updated subset of files from a previous installer.

Disadvantages

Using an installer you build with the deployment utility presents the following disadvantages:

- You must run the installer on each test system computer, which might not scale well if you use a large number of test station computers.
- The installer does not support repair.
- Users cannot select specific features to install.
- The installer requires a fixed overhead size of about 6.5 MB, which is inefficient for creating deployments that are smaller than that size.

Related concepts:

- [Relationship of NI MSI-based Installer Architecture and Microsoft Windows Installer Technology](#)
- [Building a Customized MSI-based Installer](#)
- [Using Custom Commands to Execute Third-Party Installers](#)
- [Network-Based Deployment Mechanisms](#)
- [TestStand Directory Structure](#)
- [Uninstalling a TestStand Deployment](#)

Uninstalling a TestStand Deployment

A TestStand deployment that has been installed on a test station computer will appear in the Microsoft Add/Remove Programs utility on the test station computer. Use the Add/Remove Programs utility to uninstall the distribution from the test station computer and remove all installed test system files.



Note Uninstalling a deployment using the Add/Remove Programs utility will not remove NI components that were installed with the deployment. Use the NI uninstaller to uninstall NI components from the test station computer.

Related concepts:

- [NI MSI-based Installer Architecture](#)

Building a Package Distribution with the TestStand Deployment Utility

You can use the TestStand Deployment Utility to build a package-based distribution that contains the files the test system requires to execute successfully on a test station computer, such as sequence files, support files, user interface files, the TestStand Runtime, NI run-time engines (RTEs), and NI drivers and components. Additionally, the package distribution you build with the deployment utility can perform the following tasks:

- Import hardware configuration information from Measurement & Automation Explorer (MAX)
- Create program items, such as shortcuts
- Launch a custom license agreement at install time

TestStand Deployment Utility can create the following types of package distributions:

- **Single Package**— The test system files are packaged into a single .nipkg file. Dependencies you select are referenced by the package, and must be installed on the deployment target in order to install the package.
- **Repository**— In addition to the main package, all dependent packages are also included in the deployment. A feed is generated which references all included packages. A feed is used to identify packages in NI Package Management tools, such as NI Package Manager. To install the deployment, register the feed in NI Package Manager on the target machine.
- **Package Installer**— In addition to the main package, all dependent packages are also included in the deployment. TestStand Deployment Utility also generates a standalone installer which installs all the packages on a machine. The installer will also install NI Package Manager if it is not present on the machine.



Note To create a package-based distribution for your TestStand project that includes more than a single package, NI recommends using a supported version of NI Package Builder. Refer to ***Building a Distribution with NI Package Builder*** and the ***NI Package Builder Manual*** for more information.

The Repository and Package installer modes also generate a Virtual Package (a

package with no file contents) which depends on the main package, and all package dependencies. This package is generated for future support of creating supplemental packages to add to an existing package deployment.

Related concepts:

- [Building a Package Distribution with NI Package Builder](#)

Related information:

- [NI Package Builder Manual](#)

Configuring the Package Settings

Package distribution settings are configured on the package distribution options tab. On this tab, you can choose the package name and the destination where TestStand Deployment Utility will generate the distribution.

To configure additional settings for the package, click the **Package Attributes** button.



Note The **Include Required Certificate Packages** option determines whether to include certificates packages, which reduce the number of prompts you will receive when installing the package. This setting applies only to feeds and package installers.

To Include package-based dependencies, click the **Dependencies** button. For each dependency you select, you can configure the dependency level:

- **Required (default)**— The package cannot be installed unless the dependency is on the system or can be found by NI Package Manager through a registered feed. These packages will be included in the deployment in the repository or package distribution mode.
- **Recommended**— The package can be installed without these dependencies. They are not included in the repository or package distribution mode. If the dependency package is referenced in a feed on the test station computer, the dependency will be displayed as an optional package when the deployment package is installed, and will be selected by default.

- **Optional**— The same as a recommended package, except that optional packages are not selected by default when installing the deployment package.

You can also specify a required version range for each dependency:

- **Any Version**— Do not require a specific version.
- **Specific Version**— Require the exact version number specified in the Version control.
- **Greater Than (default)**— The required package version must be greater than the version specified in the Version control. The version field is set by default to the currently installed version of the package.
- **Less Than**— The required package version must be greater than the version specified in the Version control.
- **In Range**— The required package version must be in a specified version range. When you enable this option, a second version control is available to enter the maximum version.

The following features are currently not supported in package mode:

- Creating program items in the start menu
- Registering deployed files as an activeX server
- Executing Custom Commands
- Including readme files or license agreements
- Media spanning

Building a Package Distribution with NI Package Builder

You can use NI Package Builder to build a package-based distribution that contains the files the test system requires to execute successfully on a test station computer, such as sequence files, support files, user interface files, the TestStand Runtime, NI run-time engines (RTEs), and NI drivers and components. Additionally, the package distribution you build with the deployment utility can perform additional post-installation tasks using custom execute commands.



Note You may need to update NI Package Builder to support building packages for the version of TestStand you use. Refer to the TestStand Readme and the NI Package Builder Readme for information about

supported versions.

Package Builder can create the following types of package distributions:

- **Single Package**— The test system files are packaged into a single .nipkg file. Dependencies you select are referenced by the package, and must be installed on the deployment target in order to install the package.
- **Repository**— In addition to the main package, all dependent packages are also included in the deployment. A feed is generated which references all included packages. A feed is used to identify packages in NI Package Management tools, such as NI Package Manager. To install the deployment, register the feed in NI Package Manager on the target machine.
- **Package Installer**— In addition to the main package, all dependent packages are also included in the deployment. Package Builder also generates a standalone installer which installs all the packages on a machine. The installer will also install NI Package Manager if it is not present on the machine.



Note Refer to the *NI Package Builder Manual* for more information.

Related information:

- [NI Package Builder Manual](#)

Building an Installer with Third-Party Tools

The TestStand Deployment Utility does not expose the full set of Microsoft Windows Installer technology features. You must use third-party tools to build a more customized installer that incorporates these features.

Building an installer using third-party tools requires specific knowledge of Windows Installer technology and typically requires a longer development process than using the deployment utility installer, which does not require specific knowledge of Windows Installer technology.

Advantages

Using an installer you build with third-party tools offers the following advantages:

- The installer can repair a damaged installation.
- You can build a patch installer that installs only an updated subset of the files from a previous deployment.
- The installer has a small overhead size.
- You can customize the appearance and functionality of the installer user interface to include more advanced features so users can choose which components to include at install time.
- You can add logic to decide which features to install based on what software already exists on a particular computer
- The installer can remove files it installed on the computer.
- The installer does not require test station computers to rely on a network infrastructure.

Disadvantages

Using an installer you build with third-party tools presents the following disadvantages:

- You must have specific knowledge of Windows Installer technology.
- The development process to design, build, and test the installer usually requires more time.
- You must use the deployment utility to build an installer to redistribute NI components or you must separately run the installer from the appropriate NI distribution media to install the TestStand Runtime and NI run-time engines, drivers, and components.
- You must run the installer on each test system computer, which might not scale well if you use a large number of test station computers.

Related concepts:

- [Relationship of NI MSI-based Installer Architecture and Microsoft Windows Installer Technology](#)
- [Building an Installer with the TestStand Deployment Utility](#)
- [Redistributing MSI-based Installer Products in a Deployment](#)

Network-Based Deployment Mechanisms

You can use a network drive or source code control (SCC) system to transfer test system files to test station computers, but you must also run an installer on the test station computer to correctly install the TestStand Runtime and other NI software. Because installers edit the registry, execute files, start Microsoft Windows services, call DLLs during installation, and other related tasks, copying test system files to the test station computer is not sufficient for the test system to function correctly. Because of the complexity of trying to reproduce all the actions an installer performs, NI strongly recommends that you use the NI installers or an installer you build with the TestStand Deployment Utility to install NI drivers and components and the TestStand Runtime.

Updating Test Systems Using a Network-Based Solution

When you create a new deployable image to replace an existing deployable image, delete the previous deployable image from the network drive or SCC system before you copy or check in the new version to ensure that the network drive or SCC system does not include files you removed from the deployable image.

By default, when you create a patch deployable image, the image includes only modified files. Check the patch deployable image into the SCC system or copy the image to the network drive and overwrite files in the existing full deployable image to ensure the network drive includes updated files.

If you use a network drive or SCC system to update test systems you deploy to test station computers, design the update process to manage network outages and race conditions.

Network Outages

When a test station computer cannot connect to other computers on the network during an update process, the test system might not update correctly with all the required files. For example, to change the prototype of a code module, you must update the code module and the sequences that call the code module. If the test station computer accesses and updates only the code module file but not the sequence files because of a network outage, the existing sequence files on the test station computer cannot load the new code module with the updated prototype,

which causes failures in the test system.

You can prevent partial updates by using a temporary directory on the test station computer to store all the updates before moving the updates to the test system directory. You can design the update process so a network outage during the process of updating the temporary directory generates an error and does not copy the partial updates from the temporary directory to the test station directory, which leaves the functional test system unchanged.

Race Conditions

Race conditions occur when the results of a process depend on the order in which tasks complete. To avoid race conditions when you use a network-based solution to update test systems, you must make changes to the files in the central network location before the test station computers access the files.

For example, if you change a sequence file and the related code modules on a network drive just after a test station computer loads the sequence file but before TestStand loads the code modules, TestStand loads the old sequence file and the new code modules at the start of execution. This action might lead to incorrect test results if the new code modules require limits the old sequence files do not contain or might lead to errors executing the sequence file if the code module prototypes do not match the prototypes the sequence file expects. Race conditions can also occur when sequences load code modules dynamically because old sequence files might load incompatible new code modules.

You can avoid race conditions by ensuring that the test system does not load sequences during the update process. For example, you can avoid performing updates to files in the central network location during production testing or you can transfer updates to a temporary directory and then to the test system directory only when the test system unloads all the files.

Validating Network-Based Deployments

For cases where frequent updates are needed, a Continuous Integration and Delivery (CI/CD) approach can help ensure that the latest code is available on test computers, while still ensuring the code has been properly tested and validated. With a CI/CD approach, you automate much of the procedure to build, test, and deploy updates to

test code to reduce overhead as much as possible. You can use third party tools, such as Jenkins, to implement this type of automation. For more information on using Jenkins to automate TestStand deployments, refer to the Getting Started with TestStand and Jenkins tutorial.

For more tightly controlled environments, validation requirements are typically very stringent, and therefore test system updates cannot occur frequently. For these test systems, define a separate repository or trunk where developers make changes and add new features to the test. Once the latest version of the test system has been finalized and validated, developers create a versioned branch of the trunk for use on production machines. This branch should be locked within source code control to prevent any changes to validated code. Once the new branch is completed and validated, create a new versioned branch containing the code.

Related concepts:

- [Using a Network Drive to Deploy Test Systems](#)
- [Using a Source Code Control System to Deploy Test Systems](#)
- [Redistributing MSI-based Installer Products in a Deployment](#)
- [Patching Deployments](#)

Using a Network Drive to Deploy Test Systems

You can store user interfaces, sequence files, code modules, and other TestStand components on a network drive and configure test station computers to use that network drive. You can use the deployment utility to create a deployable image that you store on the network drive. Using a network drive to update a test system updates all test station computers that use the network drive.

Consider the following issues before you decide to use a network drive to deploy or update a test system:

- **Network access**—Developers and test station computers require access to the network drive. If you cannot grant access to the network drive, you can use mirroring techniques to replicate and synchronize the network drive for the required access.
- **Network availability**—Design a fault-tolerant update process that effectively manages network outages or other situations in which a network drive is

unavailable. Network outages can stop production unless the test system can execute successfully without access to the network drive. For example, you can use a temporary directory on the test station computer to store the files you want to update and then transfer those updates to the test system directory at an appropriate time.

- **Location of files**—If the configuration of all test station computers remain identical, you can deploy and update files from the same network directory. However, test station computers might need variations based on hardware requirements, product revisions, and configuration requirements. Some test station computers might require different sequences, code modules, and driver versions to support different hardware and UUT requirements. You can use directories on the network drive to deploy or update specific versions of the test system based on hardware or UUT requirements. Test station computers also might require different configuration settings for local debugging tasks or for unique reporting requirements. You can use a network drive to distribute code modules, user interfaces, and TestStand components and use locally unique configuration files on each test station computer.



Note If you use local configuration files but modify configuration files such as GeneralEngine.cfg, Adapters.cfg, and SearchDirectories.cfg that affect all test station computers, you must edit the local copy of the configuration file on each test station computer. You can design the update process to determine which files on the test station computer to update from the network drive or use update settings files on the test station computers for greater flexibility.

- **Timing of changes**—Ensure that the update process prevents race conditions that might occur if you do not control when test station computers access files on the network drive.
- **Security**—Ensure that the network location is secure and that changes cannot be inadvertently made to the files in the test system. Because all test stations share the files, modifying the files in one location could lead to race conditions and invalidate testing.
- **Number of inbound connections**—Consult the Microsoft Software License Terms to determine the maximum number of computers that can connect to a network drive at the same time.

Advantages

Using a network drive offers the following advantages:

- For small updates, you can copy to the network drive only the files that require changes.
- Updating the contents of a network drive updates all the test station computers that use the network drive, which scales well if you use a large number of test station computers. Consult the Microsoft Software License Terms to determine the maximum number of computers that can connect to a network drive at the same time.
- You can create a deployment without having specific knowledge of Microsoft Windows Installer technology.

Disadvantages

Using a network drive presents the following disadvantages:

- Designing an update process that manages network outages and avoids race conditions requires additional development time.
- The test station computer site requires a network infrastructure of required hardware and IT resources for maintenance.
- Copying files to a test station computer is not sufficient for the test system to function correctly because you must also run installers on the test station computer to correctly install the TestStand Runtime, NI components, and required third-party components.
- You must be familiar with Microsoft Windows security issues associated with accessing and executing files from shared drive locations.
- Network drives do not keep a history of file changes. When you use the deployment utility to create a deployable image, you can save the generated deployment log to record the list of files included on the network drive.

Using a Network Drive to Deploy Patches

You can use the deployment utility to create a patch deployable image that contains all files modified since the previous full or patch deployment and replace the files on the network drive with the files in the patch deployable image folder.



Note The deployment utility creates cumulative patch deployable images. Files deleted from the deployment remain on the network drive.

Related concepts:

- [Network-Based Deployment Mechanisms](#)
- [Relationship of NI MSI-based Installer Architecture and Microsoft Windows Installer Technology](#)
- [Patching Deployments](#)

Using a Source Code Control System to Deploy Test Systems

Source code control (SCC) systems use a database to store multiple versions of files and can prevent changes from accidentally overwriting other changes. An SCC client application provides a way for users to retrieve files from or submit files to the database. You can also use the SCC client application to synchronize a set of files to retrieve the most recent versions of the files or to retrieve previous versions of the files.

In a typical SCC test system deployment architecture, you use the deployment utility to create a deployable image and submit the generated and modified files to the database on the SCC server, and test station computers retrieve the files from the SCC server. You can submit changes at any time without affecting the test system because test station computers do not receive the changes until requested. Because SCC systems use local copies of the files on test station computers, the test system can continue operation during a network outage. However, because network outages can affect SCC-based deployment mechanisms during an update, you can synchronize files to a temporary directory on the test station computer to avoid a partial update of the test system.

Additionally, you must determine when test station computers synchronize to the updated files on the SCC server. For example, a user interface can ensure that the test system has not loaded any sequences and then synchronize files at startup or before loading sequence files to automatically retrieve the most recent versions of the files.

When you use an SCC system to deploy test systems, confirm that you understand the conditions under which the SCC system overwrites files. For example, an SCC system might not automatically overwrite writeable files or files that are checked out by

another user. In this case, you might need to revert the changes and force-synchronize writeable files before you update the files to include changes you make.



Note When you create a patch deployment and select **Included** in the **File Status** control of the File Properties tab on the Distributed Files tab of the TestStand Deployment Utility, the deployment utility includes the file in the deployable image even if the file is unmodified. If the SCC system you use allows you to check in unmodified files, the next time you synchronize the file, the SCC system might overwrite any changes you made to the file on disk.

You can use the following techniques with an SCC-based deployment mechanism:

- Configure the SCC system to retrieve only the files you modified since the last time you retrieved files on the test station computer, which can increase performance over slower networks.
- Use branching features of the SCC system to concurrently develop multiple versions of a single test system because you can merge changes to large numbers of files between branches. For example, you can use a legacy system branch for bug fixes only and another branch for development tasks.
- Use separate branches for developing test systems and for storing validated test systems you deployed to test station computers, which can be helpful for separating the development and validation processes but can complicate the update process.

Advantages

Using an SCC system offers the following advantages:

- Deployment and update tasks scale well if you use a large number of test station computers because each test station computer can retrieve the most recently modified files from a central location.
- Test station computers use local copies of files from the SCC server to ensure test system operation during network outages.
- Submitting modified files does not affect executing test systems because test station computers receive changes only when requested.
- You can synchronize to previous versions of a test system if required.

- You can install files in TestStand directories, such as the <TestStand Application Data>\Cfg directory, the <TestStand Public> directory, and the <TestStand> directory.
- You can create a deployment without having specific knowledge of Microsoft Windows Installer technology.
- You can efficiently perform small updates to the deployment. The size of the files changed determines the size of the update, which does not require a large, fixed, overhead size.

Disadvantages

Using an SCC system presents the following disadvantages:

- Designing an update process that manages network outages and avoids race conditions requires additional development time. However, using an SCC system to develop an update process typically is more straightforward than developing an update process based on a network drive.
- The test station computer site requires a network infrastructure of required hardware, IT resources for maintenance, and SCC administration resources.
- Copying files to a test station computer is not sufficient for the test system to function correctly because you must also run installers on the test station computer to correctly install the SCC client application, the TestStand Runtime, NI components, and required third-party components.
- Because the source control system stores a copy of all the versions of all the files in the test system, the disk space the source control server uses could become very large.

Using a Source Control System to Deploy Patches

You can use the deployment utility to create a patch deployable image that contains all files modified since the previous full or patch deployment and add the files in the patch deployable image to the source control server, replacing existing files if required. The test station computers can synchronize with the source control server to retrieve the patched version of the deployment.



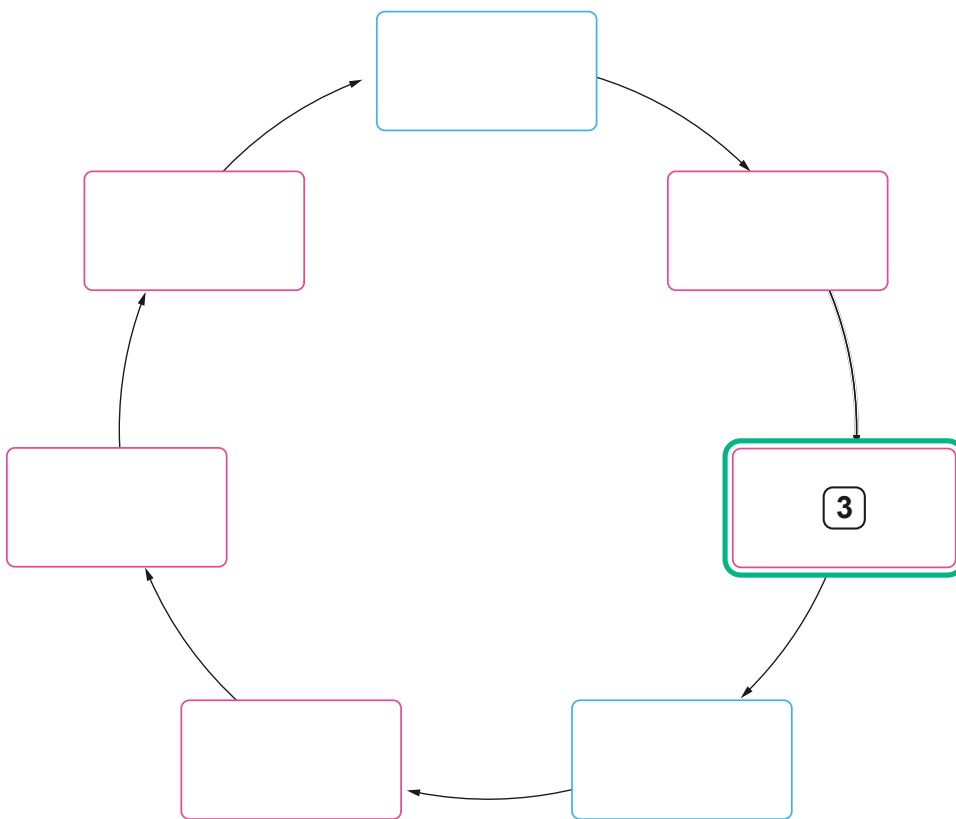
Note The deployment utility creates cumulative patch deployable images. Files deleted from the deployment remain on the test station computer. If

you want to remove these files from the test system, you must manually remove them from the source control server. Use the Status Log on the Build Status tab of the deployment utility to determine which files were removed from the test system.

Related concepts:

- [Network-Based Deployment Mechanisms](#)
- [TestStand Directory Structure](#)
- [Relationship of NI MSI-based Installer Architecture and Microsoft Windows Installer Technology](#)
- [Patching Deployments](#)

Preparing Source Components for Deployment



Use the following best practices for test system development before you create a full or patch deployment:

During Development

- **Filenames**—Use unique filenames in a test system to avoid file conflicts and to avoid including the wrong file in a deployment. Rename duplicate filenames, even if the files exist in different directories. Using files with the same name can cause sequences or the TestStand Deployment Utility to locate the wrong files, which can result in incorrect behavior. Using unique filenames ensures that TestStand locates the correct file even when you make changes to the search directories.
- **Paths**—Avoid using absolute paths because the paths might not exist on test station computers that use different operating systems, logical drive configurations, or other configuration settings, such as read-only settings. Using paths that are relative to the sequence file or to a search directory ensures that sequences locate required code modules even if you install the files to different directories on the test station computer from where the files exist on the development computer. Additionally, using relative paths avoids the use of complex file structures, which makes validating the deployment more complicated. Using absolute paths also prevents you from validating test sequences from the deployable image directory on the development computer because absolute paths point to the source files, not to the files in the deployable image directory. Additionally, the deployment utility attempts to replace absolute paths with relative paths while creating a deployment, which increases the deployment build time. To use absolute paths, you must ensure that the entire test system file and directory structures are identical on all development computers and test station computers that use the test system.
- **DLLs**—Use DLLs or LabVIEW packed project libraries to store code you share among different parts of the test system, such as code modules, or among different test systems that run on the same test station computer.
- **Workspaces**—Use a TestStand workspace during development to organize the files and static dependencies a project requires, which can simplify the deployment process. You must explicitly add dynamic dependencies to the workspace for the deployment utility to include. Additionally, you can use a workspace with a source code control system, which can aid the development and deployment process when multiple developers work on the same test system.
- **TestStand Versions**—Write all test sequences using the minimum version of TestStand the test system supports to avoid saving sequences with features or in a format the test system does not support.
- **Sequence Files**—Select **Tools»Update Sequence Files** to launch the Sequence File

Converter tool, which you can use to ensure that you save the sequence files in the test system in the format you want with the correct version of TestStand and that the sequence files use the correct versions of custom data types and step types. If the test system dynamically loads and unloads a significant number of sequence files, use the binary file format to improve execution performance.

- **Analysis Tools**—Use the TestStand Sequence Analyzer in the TestStand Sequence Editor or the stand-alone sequence analyzer application to analyze sequence files and resolve any issues the analyzer might report before you use the TestStand Deployment Utility to create a deployment.
- **Troubleshooting Tools**—Use the Debug Options dialog box, the Sequence Hierarchy window, and the TestStand File Diff and Merge utility as debugging tools to troubleshoot issues with sequence files.
- **Drivers**—As you develop code modules, maintain a list of any NI drivers or components or third-party installers with version numbers the code modules require so you can easily determine which drivers the test system requires.

Before Deployment

- If you use the Insert Code Modules context menu item in the Workspace Browser dialog box or the Insert Code Modules context menu item in the Workspace pane for any of the included workspaces, delete the code modules and insert them again to refresh any code modules that might have been added or removed.
- **Search Directories**—If the deployment includes a `SearchDirectories.cfg` file, ensure that the file does not disable the sequence file directory and workspace file directory search paths in the test system. If the deployment utility cannot find the files in the image directory through search directories, which can happen if a search directory points to the source files, the utility attempts to create relative paths from the sequence file or workspace file, which might break the deployment.

LabVIEW

- **Packed Project Libraries**—Use LabVIEW packed project libraries to store code you share among different parts of the test system, such as code modules, or among different test systems that run on the same test station computer.
- **VIs**—Save VIs in a single version of LabVIEW to reduce the time required to create a deployment. If the test system requires multiple versions of LabVIEW, use packed project libraries.
- **LabVIEW Versions**—Mass compile all VIs with the active version of LabVIEW to

avoid recompiling VIs each time you create a deployment, which slows deployment build times. Review the mass compile log file to identify any issues with compiling the VIs. The compiled code for 32-bit VIs and 64-bit VIs is not shared. If you are creating a 32-bit deployment and a 64-bit deployment, consider using LabVIEW packed project libraries or enabling the **Separate compiled code from source file** option on the General Settings page of the Project Library Properties dialog box in LabVIEW.

- **Shared Variables**—LabVIEW must create shared variables in the Shared Variable Engine before a test system can access the shared variables. The test station computer also requires a `.alias` file located in the same directory as the TestStand User Interface executable. The `.alias` file contains the IP address of the test station computer for LabVIEW code modules to use to resolve the references to the shared variable. When you call VIs in the context of a LabVIEW project, configure the LabVIEW Adapter to automatically deploy and undeploy shared variables or use the Deploy Library LabVIEW Utility step type to deploy shared variables. If any code modules in the deployment reference a shared variable outside the context of a LabVIEW project, the deployment will build successfully, but the shared variables will not be deployed on the target system, and the code modules will not run. To prevent this issue, ensure that the owning libraries of any shared variables are contained within a LabVIEW project, and that you include all such LabVIEW projects in the deployment.
- **Bound Shared Variables**—TestStand can deploy a shared variable bound to another shared variable only when you use an NI Publish-Subscribe Protocol (NI-PSP) URL to bind the shared variable to deploy to another shared variable. If you attempt to use a Deploy Library step to deploy a shared variable to a variable in a LabVIEW project, the deployment fails. Refer to the LabVIEW Help for more information about deploying shared variables and NI-PSP URLs.
- **VI Calls**—Select **Tools»Update VI Calls** to launch the Update VI Calls dialog box, in which you can verify that you compiled embedded Express VIs with the active version of LabVIEW and that all LabVIEW steps use the correct prototypes. Resolve any errors or warnings the Update VI Calls dialog box reports.

Validation

- **Development Computer**—Ensure that the test system software executes without errors on the development computer to reduce the time you spend diagnosing issues on the test station computer.

- **Installed Software**—Install the NI software the test system requires on the development computer, including the necessary patches or updates, the test system requires because the deployment utility can redistribute only the NI software installed on the computer you use to create the deployment.
 - Many NI driver installers provide different installation types that you can redistribute for development purposes or for only run-time purposes. The `Full` installation type is the most complete installation type. Installation types that support development consume more space in the installer and on disk after installation. Run-time-only installation types might exclude dependencies required to redeploy the development installation type. Distributions that NI builds, such as the NI Device Drivers DVD, always contain the full installation type for all products.
 - In most cases, you can install only one version of a hardware driver on the computer at a time. When you update drivers, validate that the latest version of the driver supports the existing hardware and executes correctly with the test system.
 - You can install multiple versions of some components, such as the LabVIEW Run-Time Engine (RTE) and LabWindows/CVI RTE, on the same computer. However, NI recommends using only one version of the LabVIEW RTE in a single test system to simplify deployment. When you build an installer with the deployment utility, the utility compiles all the LabVIEW files with the active version of LabVIEW, and you can run the VIs with the equivalent LabVIEW RTE. The LabVIEW Adapter can execute VIs faster using the LabVIEW RTE, but the LabVIEW RTE does not support debugging.
- **Source Code Control System**—Use a source code control system or other mechanism to ensure that you have the latest versions of the files the test system requires because the deployment utility includes only the files installed on the computer you use to create the deployment.
 - Verify that the required support files, such as limit files, images, and operator instructions, exist in the correct locations. Delete earlier versions of limit files to avoid including the incorrect version of the file. Receiving a File Not Found error is typically preferable to using incorrect limits. Because the deployment utility cannot detect support files that steps do not statically reference, you must manually add these types of dynamically referenced files to the TestStand workspace for the deployment utility to include them in the deployment.
 - Ensure that you include the latest build of binary files, such as executables or

DLLs for user interfaces, code modules, code module dependencies, and support files. Combining earlier and later versions of executables or DLLs might create errors that are difficult to track down. Using a dedicated build computer can minimize these types of issues.

- **Configuration Files**—If you deploy configuration files, validate that the files use the correct settings. For example, the following settings can affect execution speed and the functionality of the test system:
 - When you enable execution tracing, user interfaces can provide additional information about the execution, such as the current execution step, the status of steps, and so on, but test systems execute faster with tracing disabled.
 - Incorrect search directories can cause the test system to locate the wrong files or can prevent the test system from finding the files at all.

Related concepts:

- [Patching Deployments](#)
- [Search Directories](#)
- [Customizing Components You Deploy](#)
- [Using a TestStand Workspace File to Create a Deployment](#)
- [Manually Adding or Removing Files to or from Deployments](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Using Multiple LabVIEW Versions in a Test System](#)
- [Deploying Network-Published Shared Variables](#)
- [Resolving References to Shared Variables](#)
- [Redistributing MSI-based Installer Products in a Deployment](#)
- [Using a Dedicated Build Computer](#)
- [Configuration Files](#)
- [Deploying Stand-alone VIs](#)
- [Deploying VIs in LabVIEW Packed Project Libraries](#)
- [Deploying VIs in LabVIEW Projects](#)
- [Troubleshooting LabVIEW Code Module Issues](#)

Deploying LabVIEW Project Libraries

If the deployment includes a VI that a project library references, the deployment must include a copy of the project library, which might include files the test system does not require. However, including such non-dependent project library files in the

deployment can increase the time required to create the deployment and the disk space the deployment requires. Therefore, the TestStand Deployment Utility creates partial project libraries that include only the VIs the test system requires to reduce the time required to create deployments, the size of deployment installers, and the size of installed deployments on test station computers.

NI does not recommend using partial project libraries to deploy VIs that were previously deployed, including VIs from `vi.lib`, `instr.lib`, or `user.lib`.

Related concepts:

- [Partial Project Libraries](#)
- [Troubleshooting LabVIEW Code Module Issues](#)

Partial Project Libraries

A partial project library is a copy of a project library that uses the same qualified name as the original project library but includes references to only a subset of the VIs the original project library references.

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

Encountering Broken VIs with Partial Project Libraries

You might encounter broken VIs in partial project libraries in the following situations:

- Loading a VI that is part of a complete project library you load after loading a partial project library that does not contain the VI. For example, the complete project library `<vi.lib>\Analysis\NI_AALBase.lvlib` references `<vi.lib>\Analysis\baseonly.llb\Mean.vi` and `<vi.lib>\Analysis\baseonly.llb\Median.vi`. A deployment for a test system that requires only `Mean.vi` includes a partial project library named `Image\NI_AALBase.lvlib` that references only `Image\Mean.vi`. On a test station computer that includes the LabVIEW development system, the test system

loads `Image\Mean.vi`, which in turn loads the partial project library `Image\NI_AALBase.lvlib`. If the test system loads `<vi.lib>\Analysis\baseonly.llb\Median.vi` after it loads `Image\Mean.vi`, LabVIEW tries to load the project library for `Median.vi`, locates the partial project library named `NI_AALBase.lvlib` in memory, and tries to use the partial project library as the project library for `Median.vi`. Because the partial project library in memory does not include a reference to `Median.vi` but shares the same name as the project library that does contain `Median.vi`, LabVIEW flags `Median.vi` as broken. Closing all VIs in memory unloads the partial project library `Image\NI_AALBase.lvlib` so you can reload `Median.vi` correctly because when LabVIEW loads `<vi.lib>\Analysis\baseonly.llb\Median.vi`, it loads `<vi.lib>\Analysis\NI_AALBase.lvlib`, which includes references to `Median.vi`. Loading `Image\Mean.vi` after you load `<vi.lib>\Analysis\baseonly.llb\Median.vi` does not break `Image\Mean.vi` because `<vi.lib>\Analysis\NI_AALBase.lvlib` includes a reference to `Mean.vi` but cross-links `<vi.lib>\Analysis\NI_AALBase.lvlib` to reference `Image\Mean.vi`. You can avoid this issue by using packed project libraries that contain the VIs the test system requires. All files in a packed project library, including project libraries, use qualified names that begin with the name of the packed project library to avoid colliding with the names of files already in memory, such as partial project libraries.

- Building a LabVIEW user interface and enabling the **Remove unused members of project libraries** option on the Additional Exclusions page of the Shared Library Properties dialog box in LabVIEW. Launching the LabVIEW user interface loads the partial project libraries you create. If you configured the LabVIEW Adapter to use the LabVIEW Run-Time Engine (RTE) to run VIs and the version of the RTE matches the version of the RTE the user interface uses, LabVIEW returns broken VI errors when you execute a sequence file that includes a VI a project library references but the partial project library the user interface loaded does not include a reference to that VI. You can avoid this issue by disabling the **Remove unused members of project libraries** option and rebuilding the LabVIEW user interface.
- Loading multiple deployments that reference partial project libraries that have the same namespace but contain a different set of VIs. You can avoid this issue by building all VIs into packed project libraries with different names for each deployment to prevent name collisions among references to partial project libraries from different deployments. You can also call LabVIEW code modules in the context of different LabVIEW projects for each deployment because names of

VIs and project libraries you call in the context of a project do not collide with full or partial project libraries loaded in a separate context.

Related concepts:

- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Deploying VIs in LabVIEW Packed Project Libraries](#)
- [Troubleshooting LabVIEW Code Module Issues](#)

Troubleshooting LabVIEW Code Module Issues

Ensure that you prepared the source components as suggested because you can eliminate many issues by following the recommended guidelines and best practices. Use the following suggestions to troubleshoot issues with , , and .

Mass Compile Log

When you mass compile VIs, enable the **Log Results** option in the Mass Compile dialog box in LabVIEW and specify a destination for the log file. Review the mass compile log for broken VIs or read-only VIs.

Open broken VIs in LabVIEW and click the broken **Run** button to launch the Error list window, which includes details about the error. If no errors exist in the Error list, press the <Ctrl> key and click the **Run** button to force LabVIEW to recompile the VI, which updates the error list.

Resolve the reported errors before you include the VI in a deployment and ensure that the deployment also includes other files or drivers the VI requires.

Including Dynamically Called VIs

You must explicitly add VIs that code modules call dynamically, such as by using VI Server, to the TestStand workspace or to the directory from which you create the deployment. Additionally, you must manually add to the deployment any VIs or project libraries custom step types or other sequence elements require.

Even if a deployment builds without errors, validation tests on the test station

computer can return Error 7 with the similar text "Open VI Reference in <VI Name> VI Path: <Path to VI>" to indicate missing VIs called dynamically. Confirm that you included the dynamically called VIs in the deployment with the correct destination and verify the path of the dynamically called VIs for each dynamic call. Typically, you place dynamically called VIs in the same directory as the calling VIs.

Redeploying Previously-Deployed VIs

NI does not recommend deploying VIs that were previously deployed using the TestStand Deployment Utility. When the deployment utility packages VIs, the deployment utility includes all subVIs and creates partial project libraries that contain only the required VIs from the project libraries. If possible, use the original source files to create the new deployment.

When you attempt to redeploy VIs, the build in the deployment utility might fail with the LabVIEW error "This VI claims to be part of a library, but that library does not claim to own this VI". This error typically occurs due to conflicts between vi.lib VIs, which are included by the LabVIEW development system, and the copies of these VIs, which are included in the support VIs folder or LLB in the deployment. The deployment utility includes these copies so that the deployed files will successfully run on a machine without the LabVIEW development system installed. For more details about this error, refer to the "Encountering Broken VIs With Partial Project Libraries" section of the Partial Project Libraries topic. This issue can also occur when you deploy custom LabVIEW-based step types to a development system and then attempt to include the step types in a new deployment.

To work around this issue, you must remove the duplicate VIs and partial project libraries from the test system and relink any callers of the duplicate VIs before you create a new deployment.

Complete the following steps to remove duplicate VIs and relink their callers:

1. Create the deployment and review the VIs the deployment utility reports as duplicates in the status log.
2. Remove any duplicate VIs and partial project libraries the deployment utility reports from the SupportVIs LLB or directory. Do not remove VIs from the vi.lib folder of the LabVIEW development system.
3. Load all top-level VIs included in the previously deployed files in the LabVIEW

development system and re-save the VIs. LabVIEW prompts you to browse for any missing subVIs and relinks the VIs to the selected subVI path. In the Sequence Editor, select **Tools»Update VI Calls** to run the Update VI Calls tool to ensure that the LabVIEW steps use a valid path for the VI code module, since some steps may point to removed duplicate VIs.

4. Rebuild the deployment and verify that no library errors occur. If the errors persist, contact NI for support.

To prevent this issue in new deployments, consider enabling the **Output VIs to a packed project library** option to generate packed project libraries that contain the VIs the test system requires. All files in a packed project library, including project libraries, use qualified names that begin with the name of the packed project library to avoid conflicts with filenames already in memory, such as partial project libraries.

Related concepts:

- [Preparing Source Components for Deployment](#)
- [Manually Adding or Removing Files to or from Deployments](#)
- [Using a TestStand Workspace File to Create a Deployment](#)
- [Using a Directory to Create a Deployment](#)
- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Partial Project Libraries](#)
- [Deploying Network-Published Shared Variables](#)
- [Deploying Stand-alone VIs](#)
- [Deploying VIs in LabVIEW Packed Project Libraries](#)
- [Deploying VIs in LabVIEW Projects](#)

Using Multiple LabVIEW Versions in a Test System

NI recommends using a single version of LabVIEW to compile all the VIs for a test system. However, some situations exist that might require a test system to use multiple versions of LabVIEW. For example, you might need to update a validated test system you created in one version of LabVIEW to add a test that requires a feature available only in a later version of LabVIEW. In this case, you do not need to update and revalidate the entire test system to use the later version of LabVIEW.

You can execute code modules in different versions of LabVIEW in the same test system by using the following techniques:

- If you use the LabVIEW development system to execute code module VIs, you can force TestStand to run specific VIs in a different version of LabVIEW by enabling the **Always Run VI in LabVIEW Run-Time Engine** option in the LabVIEW Advanced Settings window. When you enable this option, TestStand always runs the VI in the LabVIEW Run-Time Engine (RTE) version that matches the version of LabVIEW in which you last compiled the VI, regardless of the option you select in the Select or Type Which LabVIEW Server to Use section in the LabVIEW Adapter Configuration dialog box.
- If you use the LabVIEW RTE to execute code module VIs, select the **Auto detect using VI version** option from the Version ring control of the LabVIEW Run-Time Engine option in the Select or Type Which LabVIEW Server to Use section in the LabVIEW Adapter Configuration dialog box. When you enable this option, TestStand always runs the VI in the LabVIEW RTE version that matches the version of LabVIEW in which you last compiled the VI.

NI recommends that you organize VIs specific to a LabVIEW version in separate sequence files and store those sequence files and related code modules in a separate directory.



Notice When you use a later version of LabVIEW to load a VI you saved in an earlier version of LabVIEW, the later version of LabVIEW recompiles the VI and marks the VI as modified. When you close the VI or exit LabVIEW, LabVIEW prompts you to save all modified VIs in memory. Accidentally saving VIs in later versions of LabVIEW might happen when you use common subVIs with VIs that require different versions of LabVIEW. NI recommends that you build common VIs into packed project libraries you compile with a specific version of LabVIEW because you can load packed project libraries only with the version of LabVIEW you used to compile the packed project library.

Deploying VIs that Require Multiple Versions of LabVIEW

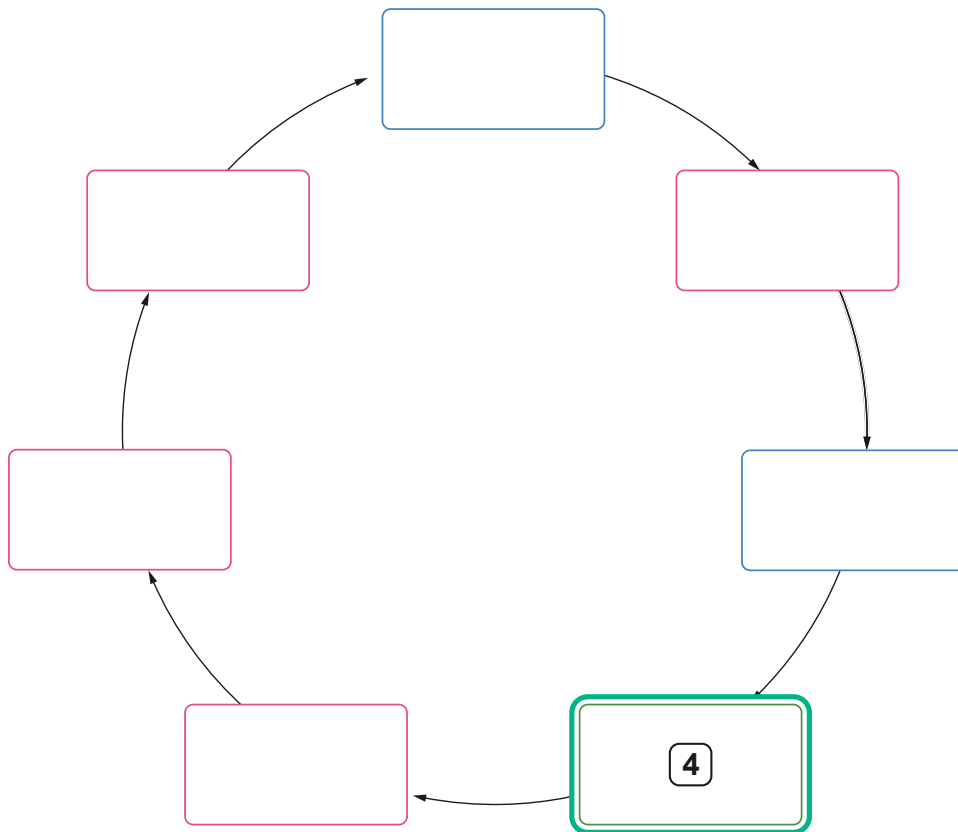
To run VIs on a test station computer that includes only the LabVIEW RTE, you must include all the VI dependencies in the deployment. The TestStand Deployment Utility analyzes and processes the VIs you include in a deployment to ensure the deployment includes the dependencies. However, the deployment utility can use only a single version of LabVIEW to process the VIs.

To work around this issue, you can create a packed project library that includes all dependent files for the deployment. Alternatively, you can create a source distribution that includes all dependent files for the deployment and enable the **Include without Processing Item or Dependencies** option in the File Properties section on the Distributed Files tab of the deployment utility for every file created in the source distribution. When you enable this option, the deployment utility does not attempt to find any dependencies of the file during analysis. Therefore, you must manually include all the dependencies in the deployment.

Related concepts:

- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Manually Adding or Removing Files to or from Deployments](#)
- [Deploying Network-Published Shared Variables](#)
- [Deploying Stand-alone VIs](#)
- [Deploying VIs in LabVIEW Packed Project Libraries](#)
- [Deploying VIs in LabVIEW Projects](#)

Customizing Components You Deploy



Use the System Source tab and the Distributed Files tab of the TestStand Deployment Utility to specify and customize the files or directories you select to include in the deployment.

Specifying Source Files

When you specify the source files for a test system on the System Source tab of the deployment utility, the deployment utility prompts you to analyze the source files before displaying the files in the Distributed Files list on the Distributed Files tab of the deployment utility. You must manually add certain types of files to the deployment. If you modify files after adding them to the deployment, click the **Analyze Source Files** button to re-analyze the files.

When you specify a TestStand workspace file on the System Source tab, the Distributed Files list on the Distributed Files tab includes the workspace file, its contents, and the dependencies, which include all the TestStand projects in the workspace, all the sequence files in each of the projects, any code modules sequence files call statically, and any other files in the project. By default, the Distributed Files list does not include code module dependencies. However, when you use the System Source tab to specify a directory from which to deploy files, the Distributed Files list on the Distributed Files tab includes all the files from that directory and the dependencies. If you use the Distributed Files tab to change the file properties of a code module dependency, add the code module dependency to a TestStand project because the Distributed Files list includes all files in a TestStand project.

Excluding Source Files

If the Distributed Files list includes files you do not want to include in the deployment, use the View Source or View Build Preview views to remove the checkmark next to the file or directory you want to exclude from the deployable image when you create a full deployment.

Customizing File Properties

Select a file or directory in the Distributed Files list and use the options in the File Properties section on the Distributed Files tab to customize how to include each file or directory in the deployable image and how the installer you build with the deployment utility for the deployable image operates on the files.

Enable the Include Without Processing Item or Dependencies option in the File Properties section on the Distributed Files tab in the following situations:

- When you want to include the source files for a user interface without modification but process code module VIs normally
- When you want to deploy VIs in a different version of LabVIEW from the code modules
- When you use checksums for strict control of which files to deploy and do not want the deployment utility to make any changes to the files
- When you want to deploy files that have already been processed, such as third-party step types that use LabVIEW VIs or a LabVIEW User Interface

Enable the Include All Files in LabVIEW Project option in the File Properties section on the Distributed Files tab to add all the files in the selected LabVIEW project to the deployment so you can more easily modify the test system you deploy. LabVIEW packed project libraries the deployment utility creates always include all the files in LabVIEW projects.

Installer Properties

The options in the Installer Properties section on the Distributed Files tab apply only when you use the deployment utility to build an MSI-based installer. These options do not affect the files in the deployable image directory.

LabVIEW Options

Use the LabVIEW VI Options dialog box to customize how the deployment utility saves VIs in the deployable image.

Using Packed Project Libraries

Enable the Output VIs to Packed Project Library option to output packed project libraries instead of VIs in a deployment, which offers the following advantages:

- Reduces the number of files in the deployment because the packed project library contains all the VIs you select.
- Occupies less disk space than VIs because packed project libraries are compressed.
- Simplifies tracking file versions, updating, and installer behavior because packed

project libraries use a version resource number, you specify.

- Ensures that you do not need to recompile calling VIs when VIs inside a packed project library change their inplaceness, or ability to reuse memory. The deployment utility enables by default the **Callers adapt at run time to Exported VI connector pane state** option on the Connector Pane State page of the Packed Library Properties dialog box in LabVIEW for all packed project libraries it creates so you can update subsets of VIs on a test station computer without mass compiling all the VIs on the test station computer.
- Simplifies upgrading test systems and preventing code duplication in test systems because packed project libraries are versioned files, which offer the following benefits:
 - Installers can correctly update the files if multiple installers that use the packed project library reside on the same computer.
 - Multiple installers can include the same packed project library, and the installer correctly installs only one copy of the file, which all code within the test system can share.

Use the Packed Project Library Options dialog box to configure options for how the deployment utility builds LabVIEW packed project libraries. You can also use LabVIEW to build packed project libraries.

Excluding VIs

The LabVIEW Development System installs `vi.lib`. Instrument drivers install VIs in `instr.lib`. Toolkits and third-party software install VIs in `user.lib`. Exclude dependencies from `vi.lib`, `instr.lib`, or `user.lib` from deployments when you deploy VIs that you expect others to edit, such as when you deploy VIs to another development computer or to a test station computer that includes the LabVIEW Development System, the TestStand Runtime, a custom user interface, and a TestStand Custom Sequence Editor License so other developers can modify and redeploy test sequences.

Including VIs from `vi.lib`, `instr.lib`, or `user.lib` in a deployment you install on a computer that includes the LabVIEW Development System can cause the following issues:

- The VIs in the deployment have the same qualified names as the source VIs in `vi.lib`, `instr.lib`, or `user.lib`, which introduces the possibility of cross-

linking.

- The deployment does not reflect changes later versions of LabVIEW make to `vi.lib`, which can result in broken VIs or VIs that do not run correctly if the VIs do not include the changes that later versions of LabVIEW require.
- Partial project libraries can cause broken VIs.

LabVIEW Optimizations

Use the Optimizations section of the LabVIEW VI Options dialog box to removed unused VI components and consolidate files projects share if you call VIs in the context of multiple LabVIEW projects.

Related concepts:

- [Manually Adding or Removing Files to or from Deployments](#)
- [Using a TestStand Workspace File to Create a Deployment](#)
- [Using a Directory to Create a Deployment](#)
- [Filtering and Excluding Files](#)
- [Using Multiple LabVIEW Versions in a Test System](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Managing Versioned and Non-Versioned Files](#)
- [Choosing Single or Multiple Deployments](#)
- [Partial Project Libraries](#)
- [Deploying VIs in LabVIEW Packed Project Libraries](#)
- [Organizing LabVIEW-Based TestStand Systems](#)

Configuration Files

TestStand stores configuration settings in `.ini` files in the `<TestStand Application Data>\Cfg` and `<TestStand Application Data>\Cfg\ModelPlugins` directories. You can add modified configuration files to a workspace to include the files in a deployment to alter the behavior of TestStand on the test station computer. If you do not include configuration files in the deployment, TestStand creates a version of the configuration files with default settings.

If you deploy a new test system that uses the TestStand 2012 or later process models to an existing system that uses the equivalent legacy TestStand 2010 process models, you must use the Legacy Model Switcher tool to migrate the report or database options to

the TestStand 2012 or later process models. To build an installer that contains the default process models, enable the **Install TestStand Runtime** option on the Installer Options tab of the TestStand Deployment Utility.

The following table lists commonly modified configuration files, specifies the information TestStand stores in those files, and specifies whether to include those configuration files in a deployment.

File	Contents	Used in Execution	Used in Editing	Include in Deployment?
<code>Analyzer.ini</code>	Settings for the TestStand Sequence Analyzer	—	✓	If you customize the default sequence analyzer settings
<code>CustomRules.tsarules</code>	Information about custom sequence analyzer rules	—	✓	If you create custom sequence analyzer rules
<code>ConnectionSource.ini</code>	Datalink list the Database and Legacy Property Loader steptypes use for configuration.	✓	✓	If the test system loads properties from databases using the Legacy Property Loader steps or uses Database step types
<code>ModelPlugins\ResultProcessing.cfg</code>	Configuration information for all result processing model plug-in instances	✓	—	If you use the TestStand process models and customized the default reporting, result logging, or database options
<code>PLPropertyLoader.ini</code>	Configuration options for the Legacy Property	—	✓	If the deployed test system uses the Import/Export Properties

File	Contents	Used in Execution	Used in Editing	Include in Deployment?
	Loader Import/ Export Properties tool Note: Legacy Property Loader steps do not use this file.			tool
ImportExportTool.ini	Configuration options for the Import/ Export Properties tool Note: Property Loader steps do not use this file.	—	✓	If the deployed test system uses the Import/ Export Properties tool
StationGlobals.ini	Definitions and values of station global variables	✓	✓	If the test system uses station global variables
Templates.ini	Templates configuration for the Insertion Palette pane in the TestStand Sequence Editor	—	✓	If you defined any templates to use in the test system
GeneralEngine.cfg	Configuration Options for stations, the Insertion	✓	✓	If you defined custom configurations for the test system

File	Contents	Used in Execution	Used in Editing	Include in Deployment?
	Palette pane, the file path history, and the type palette list			
<code>Adapters.cfg</code>	Configuration for adapters	✓	✓	If you defined custom adapter configurations for the test system
<code>SearchDirectories.cfg</code>	Configuration for search directories	✓	✓	If you defined custom search directories for the test system
<code>TypePalettes.cfg</code>	Specifies locations of type palette files	✓	✓	If you defined custom type palette files used in the deployment
<code>StartupCfg.ini</code>	Settings needed prior to engine construction	✓	✓	If you defined custom engine settings for the test system
<code>TestStandDatabaseSchemas.ini</code>	Database schemas for result logging	✓	—	If you use a custom or modified database schema to log data to a database
<code>TestStandModelModelOptions.ini</code>	Model options for built-in process models	✓	—	If you use the TestStand process models and customized the default model options
<code>ToolMenu.ini</code>	List of Tools menu items	✓	✓	If you customize the Tools menu

File	Contents	Used in Execution	Used in Editing	Include in Deployment?
				on the test station computer. Refer to Export Tools Menu Dialog Box for more information about how to create an export of a Tools menu and how to merge the export onto an existing TestStand deployed system.
Users.ini	List of TestStand users	✓	✓	If you want to configure the set of users on the test station computer
FileDiffer.ini	Configuration for the TestStand File Diff and Merge Utility	—	✓	If you want to include customized file differ settings in the test system
DatabaseViewerOptions.ini	Configuration for the TestStand Database Viewer Application	—	✓	If you want to include customized database viewer settings in the test system
ModelPlugins\Addons.cfg	Configuration for the process model plug-in add-ons	✓	✓	If you want to include process model add-ons in the test system

The following table lists commonly modified configuration files for the equivalent

legacy TestStand 2010 process models, the information TestStand stores in those files, and whether to include those configuration files in a deployment.

File	Contents	Used in Execution	Used in Editing	Include in Deployment?
<code>TestStandModelReportOptions.ini</code>	Report options for built-in process models	✓	—	If you use the TestStand process models and customized the default report options
<code>TestStandDatabaseOptions.ini</code>	Database options for built-in process models	✓	—	If you use the TestStand process models and customized the default database options

You typically do not modify the following files for use in a deployment:

- `DeploymentUtilityProtectedFolders.ini`—List of directories that cannot be modified by the TestStand Deployment Utility.
- `DeploymentUtilityRecentFiles.ini`—Most recently used file list for the TestStand Deployment Utility.
- `DeploymentUtilityOptions.ini`—Configuration options for the deployment utility.
- `DocGen.ini`—Configuration options for the Sequence File Documentation tool.
- `EngineParts.ini`—The deployment utility uses this file when you include the TestStand Runtime in an installer. Do not edit or deploy this file.
- `ExpressMenu`—Cache of the Express VI menu for the specified LabVIEW version installed on the computer. Do not deploy this file because the cache on the development computer might be different from the cache on the test station computer.
- `Filter.ini`—List of files to exclude from deployment.
- `TestStandPersistedOptions.opt`—Persisted breakpoints and watch expressions to use when no workspace is loaded.

User-Specific Application Data

TestStand stores the following sequence editor layout and user interface configuration files in the `<TestStand Local Application Data>` directory. You typically do not include these files in a deployment because users customize these files during normal TestStand use.

- `layout_current.bin`—Layout of the sequence editor the last time it was closed, including window position, size, docked state, and other settings for the panes, menus, and toolbars.
- `SeqEdit.xml`—Configuration options for the sequence editor, such as panel size and font settings used on the Insertion Palette pane and the Variables pane.
- `UserInterface.xml`—Configuration options for the user interfaces, such as window and panel sizes, font settings, and sequences to load. You can customize the data stored in this file.

Related concepts:

- [TestStand Directory Structure](#)
- [Using a TestStand Workspace File to Create a Deployment](#)
- [Station Global Variables](#)
- [Configuring Process Model Plug-ins](#)
- [Filtering and Excluding Files](#)

User Components

You can create custom components or modify installed TestStand components to change TestStand behavior on a development computer. You must include the custom components in a deployment for the test station computers to use the same modified TestStand behavior. For example, if you use a custom localized string resource file for a user interface on a development computer, you must include that file in the deployment for use on the test station computer.

The following `<TestStand>\Components` subdirectories contain files you might want to copy to a corresponding `<TestStand Public>` directory, customize, and include in a deployment:

- `Analyzer`

- Callbacks
- Icons
- Language
- Models
- StepTypes
- Tools
- TypePalettes

Related concepts:

- [Creating String Resource Files](#)
- [Components Directory](#)
- [Copying Read-Only Files to Modify](#)

Deploying Type Palette Files

Complete the following steps to deploy a type palette file.

1. Add the type palette file to the TestStand workspace you want to deploy or copy the type palette file to the directory you want to deploy.
2. On the Distributed Files tab of the TestStand Deployment Utility, click the **Analyze Source Files** button to verify that the type palette file appears in the list of files to deploy.

Type palette files are normally located in the <TestStand Public>\Components\TypePalettes directory. Complete the following steps to set the type palette file destination to this directory.

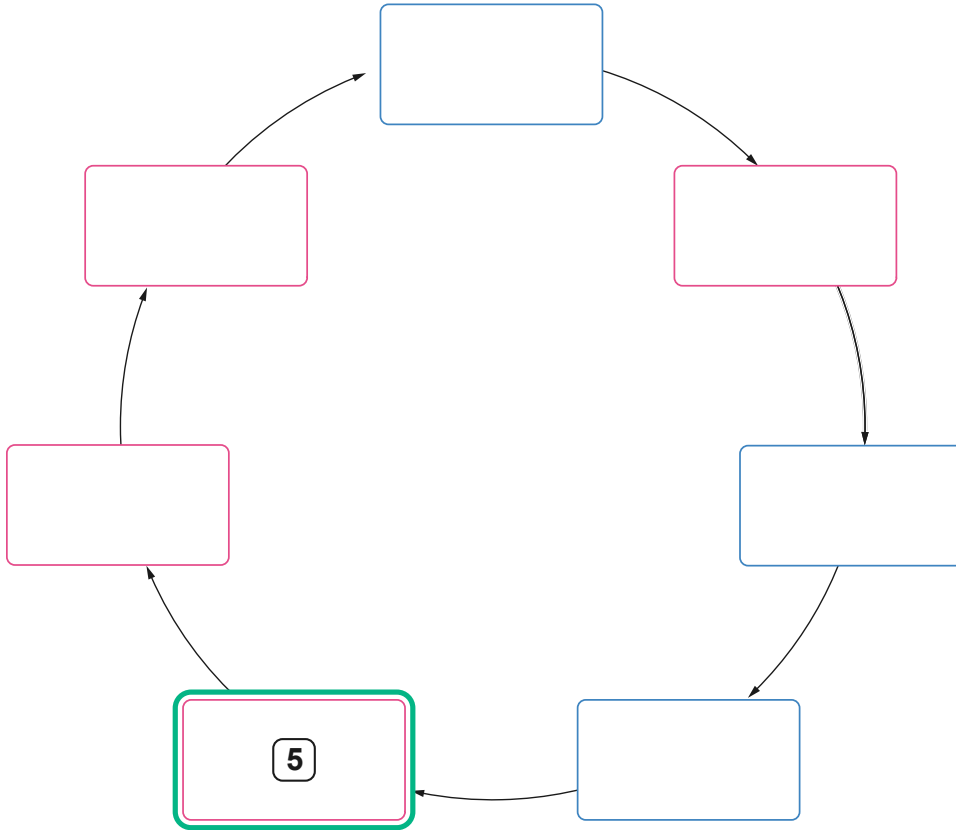
1. Select the type palette file.
2. Select **TestStand Public Documents Directory** for the **Installation Destination** option.
3. Enter `Components\TypePalettes` for the **Installation Destination Subdirectory** option.

Related concepts:

- [Type Palette Files](#)
- [Using a TestStand Workspace File to Create a Deployment](#)

- [Using a Directory to Create a Deployment](#)
- [TestStand Directory Structure](#)

Creating Deployments



When you create a full or patch deployment, the TestStand Deployment Utility completes the following tasks:

- Analyzes the test system to generate a list of source files and dependencies.
- Creates a deployable image by copying the list of files generated during analysis to a directory hierarchy under a single root directory.
- (Optional) Builds an installer or package distribution that contains all the files from the deployable image and NI drivers and other third-party components.

You can create a patch deployment at any point after the full deployment creation process. When you change to patch deployment mode, options in the deployment utility might change as well.



Note You must save the TestStand deployment specification file (.tsd) and the installer for the full deployment to create a patch deployment and installer.

Related concepts:

- [Patching Deployments](#)
- [Analyzing the Test System](#)
- [Creating a Deployable Image](#)
- [Building a Customized MSI-based Installer](#)

Using a TestStand Workspace File to Create a Deployment

You can use the TestStand Deployment Utility to create a deployment that contains all the files you specify in a workspace file.

Using workspaces and projects as the source for deployments offers the following advantages:

- Workspaces can contain files from multiple paths, which is more flexible than creating a deployment from a single root directory in the following ways:
 - You can map a directory in a workspace to a directory on the hard drive
 - You can add files from multiple directories to a single workspace directory
 - You can include only specific files from a directory on the hard drive in a workspace directory
- Workspaces contain references to files, not the files themselves. Therefore, deleting files from a workspace does not delete the files from disk.
- You can use projects as reusable, modular components that multiple workspaces can share. Defining projects by functionality can also help you easily add or remove features to or from a deployment by adding or removing the specific project to or from the workspace. For example, you can create a user interface project that includes all the files for the user interface you want to deploy.

Enable the **From TestStand Workspace File** option in the Deploy Files section on the System Source tab of the TestStand Deployment Utility and browse to or enter the path to the workspace file you want to use. When you add new files to the workspace, the deployment utility includes the new files the next time you create the deployment.



Note When you use the deployment utility to deploy only the TestStand Runtime or to deploy only the components in the `<TestStand Public>` subdirectories, you do not need to use a workspace file to organize the files for the deployment.

Creating a Workspace File for Deployment

Complete the following steps to create a workspace file.

1. Create a list of files to add to the workspace by identifying the components and required files for the test system.
2. Select **File»New»Workspace File** to create a workspace file. You must add one or more TestStand project files to the workspace.
3. Add individual files or all the files in a directory to the project file.

Avoid Inserting Code Modules in a Workspace You Use for Deployment

If you are using a workspace only to create a deployment, NI does not recommend inserting code modules that sequence files use in the workspace for the following reasons:

- Before creating a deployment, the deployment utility analyzes all the files in the workspace and includes the files that test sequences statically reference, including code modules.
- The list of code modules you insert in the workspace is a static list and does not automatically refresh when the sequence file changes. Therefore, you might deploy files that the test system no longer requires.



Note The deployment utility does not automatically include required assemblies for .NET code modules. You must add .NET assemblies to the workspace to deploy them.

Related concepts:

- [Workspaces](#)
- [Using a Directory to Create a Deployment](#)
- [Distributing Tests from a Workspace](#)

- [Deploying the TestStand Runtime](#)
- [Including Files from the TestStand Public Directory in a Deployment](#)
- [Identifying Components to Deploy](#)
- [Adding Dynamically Called Files to a Workspace](#)
- [Getting Started with TestStand](#)

Using a Directory to Create a Deployment

You can create a deployment that contains all the files in a directory. Enable the **From Directory** option in the Deploy Files section on the System Source tab of the deployment utility and browse to or enter the path to the directory you want to use. Enable the **Include Subdirectories** option to include all files in the subdirectories of the directory you select. The deployment utility automatically includes file dependencies located outside the directory you select.



Note When you use the deployment utility to deploy only the TestStand Runtime or to deploy only the components in the `<TestStand Public>` subdirectories, you do not need to use a separate directory to organize the files for the deployment.

Related concepts:

- [Deploying the TestStand Runtime](#)
- [Including Files from the TestStand Public Directory in a Deployment](#)

Analyzing the Test System

Use the System Source tab of the TestStand Deployment Utility to specify a workspace or directory that includes the components you want to deploy. The deployment utility analyzes the workspace file or directory by opening all the source files and searching for dependencies to generate a complete list of source files and dependencies to include in the deployment. For example, the deployment utility opens a workspace file and adds all the TestStand project files to the deployment, then opens each project and adds all the files in the projects to the deployment. The deployment utility continues this process until it identifies and adds all the dependencies of the test system.

Troubleshooting Deployment Analysis Errors

The deployment utility uses a log to report errors that might prevent test systems from functioning correctly, such as missing code modules. Resolve the errors and warnings on the development computer before you create or install a deployment because locating and resolving errors and warnings on the test station computer might require significant time and effort.

You can prevent many deployment issues by ensuring the test system is up-to-date and error-free by preparing the test system for deployment and by using the TestStand Sequence Analyzer to identify common problems with test sequences, such as steps that are missing code modules or code modules that TestStand cannot load.

If the deployment utility reports missing files, search the development computer to confirm the files exist in the correct directories. If the files exist on the computer but TestStand cannot find the files, use the Edit Search Directories dialog box to update the search directories, then exit and relaunch the deployment utility. NI recommends using paths relative to the workspace for TestStand projects and sequence files and paths relative to the calling sequence for all code modules to reduce the possibility of loading incorrect code modules.

If the deployment utility reports missing dependencies for code modules, you might not have installed the driver software the test system requires. Launch Measurement & Automation Explorer and expand **My System»Software** to review the NI software installed on the computer. Use the standard Microsoft Windows Control Panel facility for adding and removing programs to review the third-party drivers or other software installed on the computer.

In the following situations, the deployment utility reports that a sequence uses a file but the deployable image folder might not contain the file:

- The sequence file dynamically references the file—You must manually add dynamically referenced files to the deployment.
- The deployment utility filters files to prevent deployments from including operating system files—You can change the filtering settings using the `Filter.ini` file located in the `<TestStand Application Data>\Cfg` directory.

Related concepts:

- [Using a TestStand Workspace File to Create a Deployment](#)
- [Using a Directory to Create a Deployment](#)
- [Identifying Components to Deploy](#)
- [Preparing Source Components for Deployment](#)
- [Search Directories](#)
- [Manually Adding or Removing Files to or from Deployments](#)
- [Filtering and Excluding Files](#)
- [TestStand Directory Structure](#)

Including Source Components

Complete the following tasks before you use a TestStand workspace file or directory to create a deployment:

- Determine whether you will use multiple deployments to simplify future upgrades
- Identify the components to deploy, including files from the `<TestStand Public>` directory, a user interface and supporting files, TestStand configuration files, and NI hardware configuration information
- Prepare the components for deployment

Related concepts:

- [Using a TestStand Workspace File to Create a Deployment](#)
- [Using a Directory to Create a Deployment](#)
- [Choosing Single or Multiple Deployments](#)
- [Identifying Components to Deploy](#)
- [Including Files from the TestStand Public Directory in a Deployment](#)
- [Including a User Interface in a Deployment](#)
- [Including TestStand Configuration Files in a Deployment](#)
- [Including Hardware Configuration Information in a Deployment](#)
- [Preparing Source Components for Deployment](#)

Including Files from the TestStand Public Directory in a Deployment

The `<TestStand Public>` directory contains TestStand components you customize and copies of the TestStand User Interfaces. Enable the **From TestStand**

Public Directories option in the Deploy Files section on the System Source tab of the TestStand Deployment Utility to include these files in a deployment. If the Distributed Files list on the Distributed Files tab includes files you do not want to include in the deployment, remove the checkmark next to the file or directory you want to exclude from the deployable image in the View Source or View Build Preview view.

Related concepts:

- [TestStand Directory Structure](#)
- [User Components](#)

Including a User Interface in a Deployment

You must explicitly add user interface files to the deployment. If you are using a TestStand workspace to create a deployment, create a new project for the user interface that includes all the required files and add the user interface project to the workspace. If you are using a directory to create a deployment, ensure that the directory you use includes a copy of all the required user interface files.

Enable the **Include Without Processing Item or Dependencies** option in the File Properties section on the Distributed Files tab of the TestStand Deployment Utility for the user interface files for the deployment utility to include the user interface without modifications and without detecting dependencies because user interfaces and dependent files are designed to work together without modification.

Modifying the Built-In LabVIEW Simple or Full-Featured User Interface Examples

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

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

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

Related concepts:

- [Identifying Components to Deploy](#)
- [Using a TestStand Workspace File to Create a Deployment](#)
- [Distributing a User Interface](#)
- [Using a Directory to Create a Deployment](#)
- [Customizing Components You Deploy](#)

Including TestStand Configuration Files in a Deployment

You must explicitly add configuration files to the deployment by adding the files to a workspace or by ensuring that the directory you use to create the deployment includes a copy of the configuration files.

If you deploy to a test station computer a configuration file that is different from the configuration file you use on a development computer or if you deploy a configuration file from a directory other than the `<TestStand Application Data>\Cfg` directory, use the `Installation Destination` and `Installation Destination Subdirectory` options in the `Installer Properties` section on the `Distributed Files` tab of the `TestStand Deployment Utility` to change the installation destination of the configuration file.

None

Including Additional NI Installers in a Deployment

NI recommends that you do not install a deployment that includes NI drivers or components on the build computer, which is the computer on which you are running the `TestStand Deployment Utility` to build a custom installer for deploying TestStand

systems. If you have already installed a deployment on the build computer, NI recommends that you uninstall it. Use the NI Device Driver DVD that ships with TestStand, or a later version of the driver DVD, to reinstall the drivers/components you want to include in the custom installer the deployment utility builds.

If the version of the NI product installed on the build computer and on the test station computer do not match, the installer updates the test station computer with the version included in the installer. If the test station computer contains the later version, the installer does nothing.

If you choose to include drivers or components from a previous deployment in a TestStand installer, such as those the deployment utility builds, you might encounter the following problems:

- **The custom installer is missing features**—A custom installer that you build contains only those features that also have their dependencies installed on the build computer. For example, NI-DAQmx contains LabVIEW Real-Time drivers that depend on the LabVIEW Real-Time Module. If the LabVIEW Real-Time Module is not installed on the build computer, the custom installer will not contain the LabVIEW Real-Time drivers.
- **The custom installer removes features from the test station computer**—The test station computer contains a feature from an older NI product and that feature was not included in the installer. For example, NI-DAQmx 7.1 with the LabVIEW Real-Time Module is installed on the test station computer. The build computer has NI-DAQmx 8.0 without the LabVIEW Real-Time Module. The LabVIEW Real-Time drivers are removed from the test station computer because they depend on the LabVIEW Real-Time Module and the LabVIEW Real-Time Module was not included in the installer. NI-DAQmx 7.1 is also updated to NI-DAQmx 8.0.

Therefore, a custom installer you build might uninstall components or features on the test station computer without reinstalling updated versions of those components. As part of the installation process, the installer launches a dialog box that lists what features the installer will uninstall and allows the user to continue or quit the installation.

Related concepts:

- [Using a Dedicated Build Computer](#)

- [Calling VIs on Remote Computers](#)

Including Hardware Configuration Information in a Deployment

If the test system includes NI hardware devices, include the configuration information from Measurement & Automation Explorer (MAX) in the installer you build with the TestStand Deployment Utility for the test system to execute correctly on the test station computer. Examples of the types of configuration information you can export include NI-VISA aliases, DAQ device settings, virtual channels, IVI Logical Names, IVI Driver Sessions, NI Switch Executive Virtual Devices, and DAQ tasks you define in MAX.

Enable the **Include Hardware Configuration from Measurement & Automation Explorer** option the Drivers and Components dialog box and enter or browse to the existing NI Configuration Export File (.nce) you want to include in the deployment. You can also click the **New** button to launch the Configuration Export Wizard, which you can use to create a new .nce file. After you run the installer and reboot the test station computer, MAX imports the hardware configuration information from the .nce file after users log in to the test station computer.

Use the **Measurement & Automation Explorer Import Options** control in the Drivers and Components dialog box to specify how to merge the configuration information you include in the deployment with the configuration data defined in MAX on the test station computer.

Enable the **Display Dialog During Import** option in the Drivers and Components dialog box to launch a MAX dialog box during the installation process so users can monitor the installation and respond to prompts if necessary.



Note For more advanced hardware configuration, you can use the NI System Configuration API available in LabVIEW, LabWindows/CVI, and Microsoft Visual Studio. Use the NI System Configuration API to programmatically reboot a system, save and load system images, install and uninstall software, and obtain information about a system to be used in other applications. Refer to ni.com for more information about the NI System Configuration API.

Manually Adding or Removing Files to or from Deployments

When you specify the files to include in the deployment on the System Source tab of the TestStand Deployment Utility, the deployment utility automatically includes all the files and the statically referenced dependencies in the deployment. You must manually add to the deployment dynamically referenced files, implicitly referenced files, and . You can also filter and exclude files from the deployment.

Dynamic References

The deployment utility automatically resolves statically referenced files, which are files that have a fixed path, but cannot resolve dynamically referenced files, which are files you specify using an expression. Files you specify using an expression might evaluate to different values depending on the values of other variables during execution because TestStand evaluates expression values only at run time after specifying the values of dependent variables. You must manually add dynamically referenced files to the TestStand workspace or to the directory from which you create the deployment. Do not add files that TestStand, the operating system, or other applications install.

The following are common sources of dynamically referenced files:

- Sequence Call steps that use an expression to specify the sequence file path
- Property Loader steps that use an expression to specify the path of the file that contains the property values to use
- VIs that use the Open VI Reference function to load other VIs
- DLLs that dynamically load other DLLs
- .NET assemblies that dynamically load other assemblies

Implicit References

Implicitly referenced files are files that are not code module dependencies or referenced explicitly but that are required for the test system to execute correctly. For example, a test system can include an external calibration file in the same directory as the user interface, but no TestStand steps or code modules reference the calibration file. Common implicitly referenced files include documentation, images, and operator instructions. You must manually add implicitly referenced files to the TestStand workspace or to the directory from which you create the deployment.

Servers

The deployment utility does not automatically deploy .NET or ActiveX/COM code modules because not enough information exists to include all the necessary files to ensure that the .NET assembly or ActiveX/COM server runs correctly. For example, the deployment utility can determine that a step automates Microsoft Excel as a server using the ActiveX steps in TestStand, but the deployment utility cannot determine which files in the Microsoft Office directory Excel requires to execute correctly. To ensure that the test system executes successfully on the test station computer, use custom commands to run the installer for the code module on the test station computer or add the code module and required support files to the TestStand workspace from which you create the deployment.

NI recommends that you use an installer for the code module and its dependencies to install the required files on the test station computer instead of including the files in the installer you build with the deployment utility because the existing installer might perform additional actions required for the correct functioning of the file at install time, such as creating registry keys, running other installers, or installing and starting required Microsoft Windows services.

Related concepts:

- [Filtering and Excluding Files](#)
- [Using a TestStand Workspace File to Create a Deployment](#)
- [Using a Directory to Create a Deployment](#)
- [Using Custom Commands to Execute Third-Party Installers](#)

Filtering and Excluding Files

You can and dependencies that you do not want to include in a deployment in the following ways:

- Filter files that you want to enable or disable when you create the deployment with the deployment utility. TestStand adds filtered files to the list of distributed files but does not include the files in the deployment by default. You must manually include the files.
- Exclude files that you are sure you never want to include in a deployment. TestStand does not add excluded files to the list of distributed files, even if the files

are dependencies of included files.

For example, do not include a Microsoft Windows system DLL a TestStand step calls because deploying a Windows system DLL from a development computer that runs Windows 10/8.1/7 to a test station computer that runs Windows XP might result in operating system incompatibility issues. Do not include TestStand files the TestStand Runtime installs, such as DLLs from the `<TestStand>\Bin` directory. Install the TestStand Runtime rather than individual TestStand files to ensure you install all necessary dependencies. Do not include Microsoft Visual Studio and LabWindows/CVI files. Install these files with the corresponding run-time engine.

Filtered Files

Use file filtering based on directories and file extensions to remove dependencies you are not likely to need in the deployment. For example, the deployment utility filters executables from the `Windows` directory because those files most likely are part of the operating system on the test station computer.

The Distributed Files list on the Distributed Files tab of the deployment utility displays filtered files but does not enable the files to include in the deployment. You must manually enable filtered files to include them in the deployment. By default, the deployment utility filters the following types of files as dependencies:

- Files in the `<TestStand>\Bin`, `<TestStand>\Components`, or `<TestStand>\CodeTemplates` subdirectories
- Files in the `<TestStand Public>\Components\Tools\DeploymentUtility` directory
- Files with a `.exe` or `.dll` extension in the `Windows` or `Windows\System32` directories



Note Use caution when you include filtered files in a deployment to prevent incompatibility issues on the test station computer.

Excluded Files

Use exclusions to specify directories where dependent EXEs and DLLs should not be included in the deployment. For example, NI excludes DLLs that might cause

incompatibility issues if you include them in a deployment. Exclusions apply to code modules specified by a step and the dependencies of those code modules. The Distributed Files list on the Distributed Files tab of the deployment utility does not display excluded files to prevent you from accidentally including the files in a deployment. To include an excluded file in a deployment, you must manually add a reference to the excluded file in the TestStand workspace.



Note

- Use caution when you include excluded files in a deployment to prevent incompatibility issues on the test station computer.
- Exclusions do not apply to dependencies of LabVIEW files, including VI code modules, subVIs, or DLL dependencies of LabVIEW code modules.

Editing the Filter.ini File

You can edit the `Filter.ini` file located in the `<TestStand Application Data>\Cfg` directory to prevent including unwanted files in a deployment and to customize the files the deployment utility filters and excludes from deployments. NI recommends that you use the default values in the `Filter.ini` file and that you do not remove items from the file but instead add custom exclusions or filters.

Each of the following sections in the `Filter.ini` file contains filters, which are regular expressions that use the Match Pattern Function VI syntax.

Section	Syntax	Description
[Version]	Version="Number"	Version number of the <code>Filter.ini</code> file. The deployment utility only uses <code>Filter.ini</code> files with a later version number than the default file version number. If the version number is earlier than the deployment utility expects, the deployment utility overwrites the file with the default <code>Filter.ini</code> file. Increment the version number of the <code>Filter.ini</code> file when you want to move a <code>Filter.ini</code> file from a previous version of TestStand.

Section	Syntax	Description
[Exclude_Company_Product]	COMPANY_#="Name of company" PRODUCT_#="Name of product"	Excludes executables or DLLs with company and product name properties that match the regular expressions, which can contain wild card characters to match multiple products. For example, "Microsoft.?.?.?.?.?.?.?.?Windows" matches "Microsoft Windows" and "Microsoft Windows NT".
[Excluded_Filenames]	FILE_#="Name of the file to filter, including file extension"	Excludes DLLs with the names you specify. Use this exclusion for DLLs that do not have company or product information or when you want to use some but not all DLLs from a product.
[Excluded_Dirs]	DIR_#="Name of directory"	Excludes all EXE and DLL dependencies in the directory you specify.
[Windows]	FILTER_#="Regular expression"	Filters any file that matches the regular expression in the Windows directory.
[System32]	FILTER_#="Regular expression"	Filters any file that matches the regular expression in the Windows\System32 directory.
[TestStand]	FILTER_#="Regular expression"	Filters any file that matches the regular expression in the <TestStand> directory.
[TestStandPublic]	FILTER_#="Regular expression"	Filters any file that matches the regular expression in the <TestStand Public> directory.

Filtering Examples

Use the [Exclude_Company_Products], [Excluded_Filenames], and [Excluded_Dirs] sections to exclude DLLs by company name, product name, filename, or directory location. The `Filter.ini` file uses the company name or product name of a DLL because some applications, such as Microsoft Office, can install to any directory. The `Filter.ini` file uses filenames of DLLs because some DLLs do not define company or product names.

The following table includes examples to help you define a filter.

Use Case	Section	Syntax	Example Scenario	Example
Exclude components from a product of a certain company	[Exclude_Company_Product]	COMPANY_#="Name of company" PRODUCT_#="Name of product"	Exclude all TestStand and LabWindows/CVI files	[Exclude COMPANY Instrum PRODUCT COMPANY Instrum PRODUCT
Exclude all files that have a certain filename	[Excluded_Filenames]	FILE_#="Name of the file to filter, including file extension"	Exclude all files named readme.txt or image.jpg	[Exclud FILE_0= FILE_1=
Exclude all files from a certain directory	[Excluded_Dirs]	DIR_#="Name of directory"	Exclude all files from the Program Files directory and files from the Windows public directory	[Exclud DIR_0=" DIR_1=" (x86) " Public"
	You can also use the predefined [Windows], [System32], [TestStand], and [TestStandPublic] sections to exclude files from those directories and their subdirectories. You can also create a custom section with a name that is the absolute path of a directory to exclude files from the directory. Enter a regular expression for the filter to specify the files you want to exclude.	FILTER_#="Regular expression"	Exclude all executable files from the Windows and Windows\System32 directories and their subdirectories	[Window [System FILTER_
			Exclude all files from the <TestStand>\Bin directory	[TestSt FILTER_
			Exclude all .txt and .jpg files from the C:\Users directory and its subdirectories	[C:\Use FILTER_ FILTER_

Related concepts:

- [TestStand Directory Structure](#)

Processing Files for Deployment

The TestStand Deployment Utility copies most files unmodified from the source directory to the image directory when you create a deployment. However, the deployment utility makes changes to some LabVIEW-based code modules and TestStand sequence files to ensure the files run correctly on the test station computer.

Enable the **Include without Processing Item or Dependencies** option on the Distributed Files tab of the deployment utility to force the deployment utility to copy the files unmodified from the source folder to the image folder, which can be appropriate when you include the following types of files in a deployment:

- Files with an associated checksum
- VIs compiled in a LabVIEW version other than the active version of LabVIEW
- Files you do not want the deployment utility to modify, such as LabVIEW user interface source files or support files
- LabVIEW NXG GLL binaries

Related concepts:

- [Processing LabVIEW Code Modules for Deployment](#)
- [Processing TestStand Sequence Files for Deployment](#)

Processing LabVIEW Code Modules for Deployment

The TestStand Deployment Utility can make changes to VIs, projects, and project libraries when you create a deployment.

VIs

The deployment utility uses the active version of the LabVIEW Development System on the build computer to process code module VIs for a deployment. The deployment utility also builds a list of dependencies to include in the deployment. However, if you use the VI Server to call VIs dynamically, you must manually add the VIs to the deployment because the deployment utility cannot automatically include VIs you call dynamically.

Code module dependencies can include subVIs, VIs in `vi.lib`, non-operating system

DLLs, external subroutines, and run-time menus. The deployment utility includes these dependencies in the deployment to ensure that the code modules execute correctly on test station computers. For example, test station computers typically use only the LabVIEW Run-Time Engine (RTE), which does not include `vi.lib`. Deploying code module VIs that call `vi.lib` VIs to a test station computer with only the LabVIEW RTE installed without using the deployment utility to first process the code module VIs results in broken VIs on the test station computer.

You can also configure the deployment utility to change other aspects of VIs, such as changing the destination of code module VIs and subVIs. Use the LabVIEW VI Options dialog box to modify the behavior of VIs in the deployment in the following ways:

- Output VIs to packed project libraries
- Remove block diagrams for all VIs
- Apply a new password to all VIs
- Check for broken VIs
- Remove unused polymorphic VIs
- Consolidate VIs that different LabVIEW projects share
- Exclude dependent VIs located in `vi.lib`, `user.lib`, or `instr.lib`

LabVIEW Projects

If a TestStand sequence file executes VIs in the context of a LabVIEW project, the deployment utility applies the following changes to the copy of the LabVIEW project it includes in the deployment:

- Updates file path references in the LabVIEW project to correspond to the deployable image file paths
- Removes files from the LabVIEW project that the test system does not use
- Converts folders that LabVIEW automatically populates into static folders

LabVIEW Project Libraries

The deployment utility minimizes the size of the deployable image by creating a partial project library that does not include VIs from LabVIEW project libraries the test system does not use.

VIs that Call DLLs

You can use the deployment utility to build an installer that contains VIs that call into DLLs with dependencies, such as other DLLs or configuration files. Alternatively, you can call DLLs from a TestStand step.

For DLLs that VIs and TestStand steps call, the deployment utility automatically includes in the deployment the DLLs and the statically referenced dependencies of the DLLs. The deployment utility does not automatically include dynamically referenced dependencies or implicitly referenced dependencies. You can manually add the DLLs, the dynamically referenced dependencies, and the implicitly referenced dependencies to the TestStand workspace or to the directory from which you create the deployment to maintain the same file and directory structure on the test station computer so the DLL can correctly locate the dependencies in the test system you deploy.

Related concepts:

- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Using a Dedicated Build Computer](#)
- [Manually Adding or Removing Files to or from Deployments](#)
- [Customizing Components You Deploy](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Partial Project Libraries](#)
- [Deploying 32-bit TestStand and 64-bit TestStand Systems](#)
- [Deploying VIs in LabVIEW Packed Project Libraries](#)
- [Deploying VIs in LabVIEW Projects](#)
- [Deploying Stand-alone VIs](#)
- [Troubleshooting LabVIEW Code Module Issues](#)

Processing LabVIEW NXG Code Modules for Deployment

TestStand Deployment Utility supports deploying sequence files which call LabVIEW NXG code modules. LabVIEW NXG steps can be configured in the following ways:

- **Project and GLL are Both Specified**— These steps can execute using the LabVIEW NXG development system to execute the VI directly, or using the LabVIEW NXG Run-Time Engine to execute the VI within the GLL. In this case, TestStand enforces that the specified GLL must match the GLL output configured in the project and

component in LabVIEW NXG. When deploying these types of steps, TestStand Deployment Utility automatically regenerates the GLL from the project before deploying to ensure that the latest code is used.

- **GLL Specified, Project Not Specified**— These steps must be executed using the LabVIEW NXG Run-Time Engine to execute the VIs built into the GLL. When deploying these types of steps, the GLL file is deployed as-is because no source files are present to rebuild it.
- **Project Specified, GLL Not Specified**— These steps can execute using the LabVIEW NXG Development System to execute the VI directly. Deployment is not supported for these steps by TestStand Deployment Utility. An error is generated indicating that a GLL is required.



Note If a LabVIEW NXG file is included in a deployment but not referenced by the sequence file, TestStand Deployment Utility will not process the file and will include it as-is.

Configuring LabVIEW NXG Files

After analyzing a sequence with LabVIEW NXG code modules, the referenced GLLs are displayed in the distributed files tab. You can configure the destination for the GLL file. Like other code module types, TestStand Deployment Utility ensures that the sequence file is updated to reference the correct files if the files are deployed within the same base destination as the calling sequence file.

To include LabVIEW NXG source files in the deployment, click **LabVIEW options**, and select the LabVIEW NXG options tab. Enable the **Include LabVIEW NXG Source Files** option. With this option selected, LabVIEW NXG projects, components, and VIs will also be displayed in the distributed files tab. Only referenced files are displayed in the distributed files tab, but all files included in the project will be included in the deployment. You can configure the destination for the source project, but not for contained components and VIs; these files will be deployed to a location relative to the project.

Deploying the LabVIEW NXG Run-Time Engine in an MSI-based Installer

Because the LabVIEW NXG Run-Time Engine is a NI package it is not directly available as a dependency in an MSI-based installer. However, if you enable the **Automatically**

Include Required Installers option in the Drivers and Components dialog, TestStand Deployment Utility will include the Run-Time Engine package within the installer folder. When the installer is executed, it will invoke NI Package Manager to install the Run-Time Engine package.

Processing TestStand Sequence Files for Deployment

The TestStand Deployment Utility replaces absolute file path references with relative file path references because the location of files on a development computer might be different from the paths on the test station computer. For example, on the development computer, you might save files in the `C:\Dev\TestSystem` directory but install files on the test station computer in the `C:\<Program Files>\TestSystem` directory.

The deployment utility updates absolute path references in sequence files to use relative path references that initiate from one of the following search directories:

- Current sequence file directory
- `<TestStand>`
- `Windows\System32`
- `Windows`
- `<TestStand Public>`

When the code module file you reference does not exist in one of these search directories, the deployment utility does not update the absolute paths. Therefore, the test sequence might not execute correctly on the test station computer.

The deployment utility also updates paths to code modules if you change the destination of code module files. For example, if a code module and its calling sequence file exist in the same source files directory but you change the destination directory for the code module to a subdirectory, the deployment utility updates the sequence file to use the new path to the code module when you create the deployment.

When you output VIs to packed project libraries, the deployment utility updates all appropriate steps in the sequence file to use the correct VI path and namespace.



Note The deployment utility does not modify custom step types. Instead, the deployment utility reports a warning on the Build Status tab when custom step types use absolute paths.

Related concepts:

- [Search Directories](#)
- [TestStand Directory Structure](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Deploying 32-bit TestStand and 64-bit TestStand Systems](#)

Processing .NET Code Modules for Deployment

If you are using TestStand 2024 Q4 or later, you must manually add .NET code modules and support files, like `runtimeconfig.json` and `deps.json` files, when creating deployments that call .NET assemblies.

You must manually add .NET code module files to your deployments for both:

- deployments that call .NET code using the .NET Adapter, and
- deployments that include other files, like VIs, that call .NET code during their operation.



Note If your deployment includes LabVIEW VIs developed in LabVIEW 2025 Q1 that call .NET assemblies, these VIs automatically include the called .NET assemblies as deployment dependencies. .NET dependencies called by included .NET assemblies are not included automatically - you must manually add these dependencies to your deployments.

Creating a Deployable Image

The TestStand Deployment Utility collects all the files the deployment requires into one directory called the deployable image, which contains the files and statically referenced dependencies you specify on the System Source tab. Creating a deployable image with the deployment utility helps you verify that you include the required files and process the files before you build an installer or use a network drive or source

code control system to deploy the files.

When you create a patch deployment, the deployment utility removes files from the patch deployable image that have not changed since the previous full or baseline patch deployment image.

Troubleshooting the Deployable Image

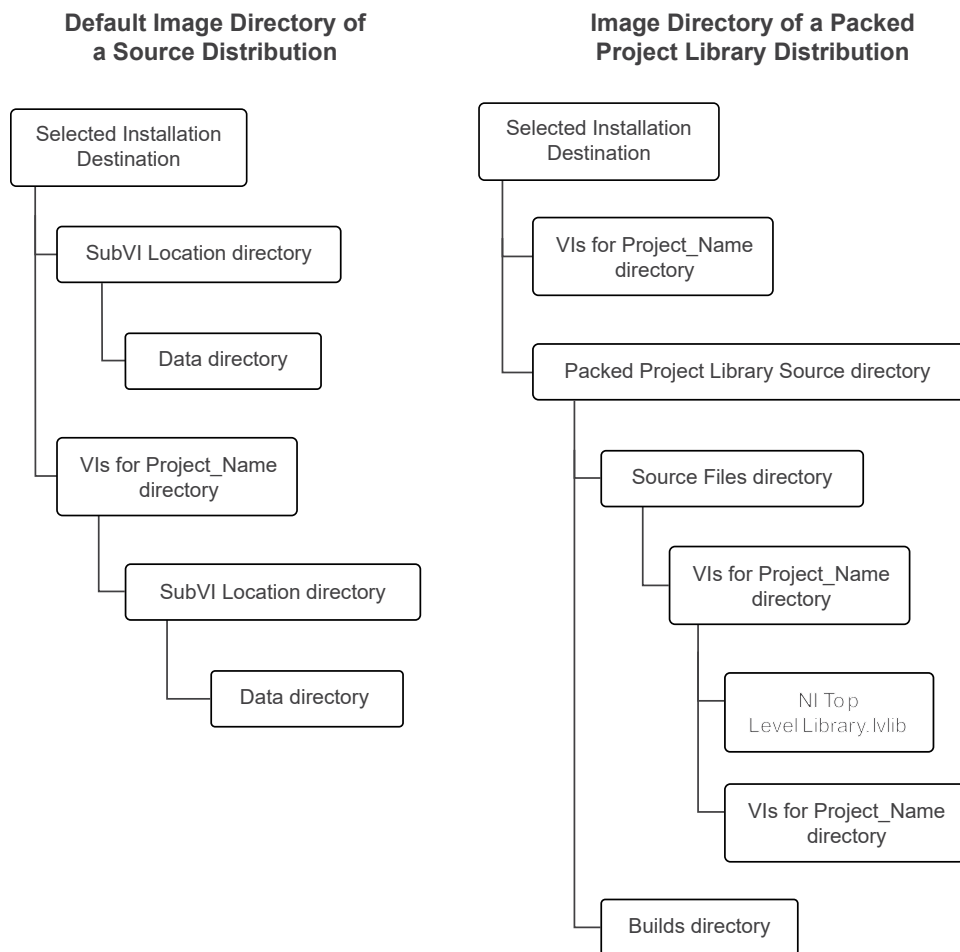
Use the deployment utility status log to diagnose any potential problems with the deployment. If the log has any errors or warnings, the help page displays information about the steps required to fix the issue.



Note The purpose of information messages, such as the message about expressions that specify code module paths, is to alert you to common mistakes during deployment so you can take the required actions to prevent the associated problems.

Directory Structure Differences between Source Distributions and Packed Project Library Distributions

You can include in the deployable image the source files necessary to rebuild packed project libraries, but doing so places the packed project library source files in a different directory structure from the default source distribution directory structure, as shown in the following figure.



When you create a source distribution that contains packed project libraries, the deployment utility includes dependencies of the Packed Libraries, including other packed project libraries and shared libraries (DLLs), in the SubVI Location directory you specify in the LabVIEW VI Options dialog box.

When you create a packed project library distribution, the deployment utility automatically includes dependencies in the same top-level directory as the packed project library.

If you previously deployed a source distribution and then switch to using a packed project library distribution, the directory structure of the new deployment changes but the functionality remains the same. Typically, you do not need to make changes to the new deployment unless you are using a source code control system for distribution and rely on consistent locations of files for revision history information.

Default Image Directory of a Source Distribution

The default image directory of a source distribution includes the following directories and files:



Note

- The deployment utility can create additional directories depending on the destination subdirectory of the top-level files in the source distribution. Use the Distributed Files tab to change the destination subdirectory of top-level files.
- Patch deployments use the same image directory structure, and the deployment utility automatically removes any files the patch deployment does not require.

- **Selected Installation Destination**—Contains top-level files, such as VIs, sequences, LabVIEW projects, TestStand workspaces, and TestStand projects. Select a file on the Distributed Files tab of the deployment utility and use the Installation Destination option to specify this directory. Use the Installation Destination Subdirectory control to specify a subdirectory.
 - **SubVI Location directory**—Contains direct dependencies of the top-level VIs, including subVIs, DLLs, and libraries of the subVIs. You specify this directory in the SubVI Location option in the LabVIEW VI Options dialog box.



Note If a direct dependency of a top-level VI is also a top-level file, such as when you call a DLL from a VI and also call the DLL from a step in a sequence, the deployment utility places the file in the path you specify in the Installation Destination Subdirectory control. The deployment utility also correctly links the VIs and sequences that call the file.

- **Data directory**—Contains indirect dependencies of the top-level VIs, including libraries of top-level files and classes.
- **VIs for Project_Name directory**—Contains top-level files for the project. If you consolidate projects, the deployment utility creates only one VIs for

`Projects` directory in which to store all the top-level files for all projects.

- **SubVI Location directory**—Uses the same name and structure as the top-level SubVI Location directory.
- **Data directory**—Uses the same name and structure as the top-level `Data` directory.

Image Directory of a Packed Project Library Distribution

The image directory of a packed project library distribution includes the following directories and files:



Note Patch deployments use the same image directory structure, and the deployment utility automatically removes any files the patch deployment does not require.

- **Selected Installation Destination**—Contains top-level files, such as packed project libraries, sequences, LabVIEW projects, TestStand workspaces, and TestStand projects. This directory also contains packed project library dependencies, such as DLLs called from VIs in the packed project library. Select a file on the Distributed Files tab of the deployment utility and use the Installation Destination option to specify this directory. Use the Installation Destination Subdirectory control to specify a subdirectory.
 - **VIs for Project_Name directory**—Contains top-level files for the project. If you consolidate projects, the deployment utility creates only one `VIs for Projects` directory in which to store all the top-level files for all projects.
 - **Packed Project Library Source directory**—Contains one LabVIEW project for each packed project library in the deployment and one readme file for each project. (LabVIEW 2012 SP1 or earlier) After the projects have been built in the Packed Project Library Source directory, this directory contains the packed project libraries generated by the projects with the same directory structure as the selected installation destination.
- **Source Files directory**—Contains directories for each project.
 - **VIs for Project_Name directory**—Contains all the files added to the top-level library in the project, including top-level files, libraries of the top-level files, and other VIs in the libraries of the top-level files.

- `NI Top Level Library.lvlib`, which is the top-level library for the deployment.
- **Support VIs directory**—Contains dependencies of the top-level VIs in the project, including subVIs, DLLs, and libraries of the subVIs.
- **Builds directory**—(LabVIEW 2013 or later) After the projects have been built in the Packed Project Library Source directory, this directory contains the packed project libraries generated by the projects with the same directory structure as the selected installation destination.

Related concepts:

- [Identifying Components to Deploy](#)
- [Preparing Source Components for Deployment](#)
- [Choosing a Deployment Mechanism](#)
- [Patching Deployments](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Deploying 32-bit TestStand and 64-bit TestStand Systems](#)
- [Deploying VIs in LabVIEW Packed Project Libraries](#)
- [Installer Issues for 32-bit TestStand and 64-bit TestStand](#)

Choosing Single or Multiple Deployments

Typically, you can expect to update a deployed test system to fix bugs or to add more tests over the lifetime of the test system. The initial configuration of the test system determines the size of the installer and the effort involved to build the installer to update the test system.

You can create a or for a test system. Each TestStand deployment specification file (`.tsd`) contains the information to create a single deployment. To create multiple deployments with different contents, define one `.tsd` file for each deployment.

Single Deployment

If you use a single deployment that contains all the required components for the test system, when you build a patch installer, the deployment utility must analyze and create a deployable image of the entire test system to determine the changed files to include in the patch deployment. Using this technique can increase the time required to build the patch installer to update a test system.

Multiple Deployments

You can use multiple deployments to divide the parts of the test system into groups that you most likely will upgrade at the same time. For example, you can create a deployment that contains the TestStand Runtime and another deployment that contains the rest of the test system because the TestStand Runtime is a large component that does not usually change when you update a test system. One common strategy is to use one deployment for the TestStand Runtime, another deployment for TestStand framework files, such as user interfaces and process models, and a third deployment for the rest of the test system, including test sequences and code modules. You can also create product-specific deployments that include only the test sequences and code modules required to test that product, but using this technique can become complex.

Include Shared Code in a Separate Deployment

Use separate deployments for shared code to ensure that code modules execute without errors after installation and to ensure that deployments execute correctly when you uninstall shared files.



Note Because of the issues that can result from using installers you build with the TestStand Deployment Utility to deploy shared files, NI recommends that you use only LabVIEW packed project libraries to deploy shared code and avoid sharing other types of files, such as VIs, DLLs, LLBs, sequence files, and so on, among multiple deployment utility installers.

Ensuring Shared Code Executes without Errors after Installation

Updating a shared file that multiple deployments use can affect code modules in the other deployments that depend on the shared file, especially when you use LabVIEW code modules. For example, changes to the inplaceness of a subVI require that you recompile all VIs that call the subVI. Determining which deployed VIs require recompilation can be difficult because you cannot easily detect inplaceness changes or find all the callers of a VI. Additionally, removing VIs from a shared project library can affect the VIs in other deployments.

For example, consider that you deployed Caller VI and Called VI. After deployment, you

make a change in the Called VI that changes memory allocation, such as changing the data wired to an indicator to a copy of the data of another indicator, and then save the Called VI. When you attempt to run the Caller VI from the test system using the LabVIEW Run-Time Engine, the VI appears broken but loads correctly in the LabVIEW development system despite including a coercion dot.

For LabVIEW code modules, build packed project libraries and enable the **Callers adapt at run time to Exported VI connector pane state** option on the Connector Pane State page of the Packed Library Properties dialog box in LabVIEW. If you do not use this technique and you want to change VIs in a deployed test system, you must redeploy all the VIs in the test system to prevent run-time errors when LabVIEW recompiles VIs to account for inplaceness changes.

Uninstalling One Deployment Can Affect Remaining Deployments that Use Shared Files

When multiple deployments use installers you built using the deployment utility to install the same set of shared files and you use one of the uninstallers to remove a deployment, the deployment utility uninstaller removes the shared files from the test station computer, which might affect the remaining deployments that require the shared files. You must uninstall the broken deployment and reinstall it to work around this issue.

You can use LabVIEW packed project libraries to deploy shared files because the deployment utility installer marks the packed project library as a shared file, which the operating system reference counts correctly and does not uninstall the file until you uninstall all installers that require the shared file.

Related concepts:

- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Upgrade Codes That Affect MSI-based Installer Behavior](#)

Using a Dedicated Build Computer

Identifying problems in a deployment as soon as possible can help avoid errors on test station computers. Testing the deployment image on the development computer you

use to create the deployment is helpful because the development computer includes the required tools to fix the issue and rebuild the deployment. You can also use a dedicated build computer on which you only create deployments and avoid any development tasks.

Using the same computer to develop and deploy a test system presents the following disadvantages:

- The deployment might include files in an incomplete state. A common development practice of having multiple versions of a file to test different approaches can result in including one of these files in the deployment by mistake.
- Files that were not submitted to the source code control (SCC) system do not produce errors during deployment because the files exist on the development computer. Additionally, adding new developers or moving to a new development computer might result in lost work.
- The deployment might include files that reference dependencies using absolute paths. If these dependencies are installed to a different path, errors may occur on a deployed system. These errors will not occur on the development system since the dependencies exist at the absolute path.

Using a dedicated build computer offers the following advantages:

- Files synchronized using an SCC system on the build computer can ensure that the files are in the correct state.
- The build process can report as an error files not submitted to the SCC system.
- You can automate the build process to run without negatively affecting developer productivity.

Virtual Machine or Image

If you do not have a separate computer available to dedicate to building deployments, you can use a virtual machine or a disk image the same way you use a dedicated build computer.

Building a Customized MSI-based Installer

Use the TestStand Deployment Utility to build an MSI-based installer for the deployable image or patch deployable image.



Note You must save the MSI-based installer for the full deployment to build an MSI-based installer for the patch deployment.

Use the MSI-based Installer Options tab of the deployment utility to customize the MSI-based installer as a whole, and use the Distributed Files tab of the deployment utility to customize the installation of individual files. You can include the TestStand Runtime and MSI-based installers for NI hardware drivers and other software components. An MSI-based installer you build with the deployment utility automatically requests a reboot when any of the NI drivers or software components you redistribute require a reboot, such as the TestStand Runtime.



Note Refer to <National Instruments>_Legal Information for information about including legal information in MSI-based installers built with NI products.

Upgrading Previous Installations

Use the following table to determine which technique to use to upgrade a previous installation.

Situation	Action
New MSI-based installer updates previous MSI-based installer	• Create a full deployment that can update an existing installation or that can install to test station computers that do not have the previous installation. To upgrade using a full deployment, use the Advanced MSI-based Installer Options dialog box to increment the installer version but do not modify the upgrade code. Generate a new upgrade code to install a full deployment side-by-side with a previous version of the deployment. • Create a patch deployment that can only update a previous installation. Using this technique can significantly reduce the size of the MSI-based installer. To upgrade using a patch deployment, use the Mode tab of the deployment utility to create a patch deployment, which automatically updates the version and upgrade code to install the patch deployment. When you create a cumulative patch deployment, the deployment utility generates a new upgrade code so you can

Situation	Action
	install patch deployments on the test station computer. If you create a patch deployment from a baseline patch deployment, the deployment utility uses the upgrade code of the baseline patch deployment.
Files already exist on test station computer	Review the scenarios in which an MSI-based installer overwrites an existing file.
You want to override the default MSI-based installer behavior	Enable the Force File to Install option on the Distributed Files tab.  Notice When you enable the Force File to Install option, an MSI-based installer can replace a newer version of a file with an older version of a file, which can be problematic for DLLs because components on the computer might rely on functionality available only in the newer version of the DLL and might not execute correctly when the MSI-based installer downgrades the DLL.

Troubleshooting MSI-based Installers You Build with the TestStand Deployment Utility

Use the following techniques to troubleshoot building MSI-based installers with the TestStand Deployment Utility:

- Ensure the hard drive has sufficient disk space for building the MSI-based installer. For example, including the TestStand Runtime requires about 450 MB of disk space.
- Clear the product cache if you are including a NI component or driver in the MSI-based installer to remove out-of-date or corrupt files.
- Review the Status Log on the Build Status tab of the deployment utility for more information about MSI-based installer build errors. Right-click the Status Log and select **Group By»Severity** or use the column headers to filter and display only errors and warnings to help you locate possible problems with the MSI-based installer. Click the **Save Log** drop down button and select **Save Technical Support**

Report to save a copy of the log that NI can use the log files the deployment utility generates to help you diagnose MSI-based installer-related issues.

Installing TestStand Silently

You can install TestStand without viewing any installation dialog boxes. Refer to the NI support article 4CJDP38M, Automating the Installation of a Single Installer, on the NI website for information about installing NI software silently.

Related concepts:

- [Patching Deployments](#)
- [Redistributing MSI-based Installer Products in a Deployment](#)
- [Clearing the Product Cache](#)
- [Deploying 32-bit TestStand and 64-bit TestStand Systems](#)
- [Installer Issues for 32-bit TestStand and 64-bit TestStand](#)

Relationship of NI MSI-based Installer Architecture and Microsoft Windows Installer Technology

Use Microsoft Windows Installer technology in conjunction with the NI MSI-based installer architecture to redistribute NI drivers and other software components, such as run-time engines, with a deployment.

A Windows Installer installs, uninstalls, and repairs applications. For a single application, a Windows installation package includes a Windows Installer database file (.msi), which contains all the information a Windows Installer requires to install or uninstall an application and to run the setup user interface.

You can use Windows Installer technology to accomplish the following tasks:

- **Support unattended program installation**—You can script a program installation according to administrator instructions.
- **Help prevent certain forms of inter-program conflicts**—The MSI-based installer enforces installation rules that help prevent conflicts that can occur when an installation or removal operation makes updates to or deletes a DLL an existing program shares.
- **Diagnose and repair corrupted programs**—You can determine whether an

installed program has missing or corrupted files and repair the program by reinstalling only the missing or corrupted files.

- **Reliably remove existing programs**—The MSI-based installer can remove any program it previously installed, including all associated registry entries and program files, except for files other installed software shares.
- **Support on-demand installation of program features**—The MSI-based installer can initially install only a minimal subset of program features. You can automatically install additional components the first time you use a feature that requires the components.
- **Restore the original computer state if an installation failure occurs**—A Windows Installer records all changes made to the computer during the program installation process and can restore the computer to its initial state, also known as a rollback.

Related concepts:

- [NI MSI-based Installer Architecture](#)
- [Redistributing MSI-based Installer Products in a Deployment](#)

Microsoft Windows Merge Modules

Microsoft Windows Installer technology uses merge module (.msm) files as a standard method for developers to specify logic among applications and to deliver shared code, files, resources, registry entries, and other components. Merge module files contain the database tables Windows Installer database files require to install a component. You cannot install a merge module file directly. You must incorporate the merge module file into a Windows Installer database file. Because of this requirement, using merge module files with Windows Installer database files to install interdependent NI products can be problematic for the following reasons:

- Multiple merge module files can contain conflicting table entries, which prevents you from incorporating the merge module files into a single Windows Installer database file. Using the NI MSI-based installer architecture, NI distributions avoid this issue by using parts based on Windows Installer database files, which use independent table entries.
- Merge module files duplicate the content for every package that includes the merge module files, which can increase the amount of space needed for complex distributions. Using the NI MSI-based installer architecture, NI distributions avoid

this issue by reusing parts in multiple products.

Related concepts:

- [Relationship of NI MSI-based Installer Architecture and Microsoft Windows Installer Technology](#)
- [NI MSI-based Installer Architecture](#)

NI MSI-based Installer Architecture

Microsoft Windows Installer technology works well for single, independent applications, but many NI products are interdependent and share sets of features. Therefore, NI developed an MSI-based installer architecture to accomplish the following tasks:

- Install, uninstall, or upgrade products independently or as part of a larger installation
- Define dependencies among products
- Provide a common user interface so all installations or uninstallations of NI products appear as a single operation, which is necessary because each NI product uses a series of smaller MSI-based installers

The NI MSI-based installer architecture uses parts, which represent the smallest unit the NI MSI-based installer requires to track by version or the largest indivisible group of features. Parts are Windows installation packages that contain one or more Windows Installer database files that include additional custom tables NI defines.

A product consists of one or more parts and an XML file that describes the product. You can include a part in several products to share common features among products. The XML file lists the product requirements and properties, such as the product name, description, and whether that product is redistributable, which means that you can include the product in other MSI-based installers. For example, you can include the LabWindows/CVI Run-Time Engine in an MSI-based installer you build with the TestStand Deployment Utility.

A distribution includes one or more parts and required support files for all the features of a product. Running the installer executable from the distribution media might

display a list from which you can select features to install. Customizing this list changes the features a part installs or changes the set of parts to install. During installation, the setup application iterates through each part that contains the features you selected to install. If a part is not already installed or if the part is a later version than the part already installed on the computer, the setup application installs the part. If the part already installed on the computer specifies a version that is a later or equal version to the version of the part in the MSI-based installer, the MSI-based installer does not install that part. The MSI-based installer user interface represents the series of part installations as a single, unified installation by showing a progress bar for the current part and an overall progress bar for the installation of all the parts.

When you remove NI software, the MSI-based uninstaller user interface lists the products and patches installed on the computer from which you can select the products to remove. Before removing a product, the NI MSI-based uninstaller confirms that no other products on the computer require the product parts you want to uninstall.

When you modify features of a product, the NI MSI-based installer installs the parts required for the features you add and uninstalls parts for the features you remove. Before removing a part, the NI MSI-based uninstaller confirms that no other products on the computer require the parts you want to uninstall.

Related concepts:

- [Relationship of NI MSI-based Installer Architecture and Microsoft Windows Installer Technology](#)
- [Redistributing MSI-based Installer Products in a Deployment](#)
- [Uninstalling a TestStand Deployment](#)

Redistributing MSI-based Installer Products in a Deployment

You can install a product and then include that product in an MSI-based installer you build to redistribute the product with a deployment. NI provides product distributions for you to include in the deployments you create. By redistributing products in a single, customized MSI-based installer that includes only the features of the product the test

system requires, you can streamline the installation process so users can avoid separately installing multiple products with specialized feature sets. Redistributing products in a single MSI-based installer also avoids the need to distribute multiple distribution media along with the test system you deploy.

Redistribution Using Merge Modules in TestStand 3.0 and 3.1

In TestStand 3.0 and 3.1, you could use the deployment utility to redistribute the TestStand Engine by recreating the MSI-based installer from a Microsoft Windows Installer database file and a large number of merge modules. However, using this technique caused the following issues:

- You could not extract merge modules once you merged the files into a Windows Installer database file. Therefore, the TestStand MSI-based installer had to include copies of all the merge modules needed to rebuild itself, which nearly doubled the size of the TestStand MSI-based installer.
- Merging a large number of merge modules and building MSI-based installers was time-consuming and did not scale well for large distributions with multiple products, such as multiple NI drivers.
- This technique supported only a few NI products. Adding support for other NI products was difficult because no standard existed for obtaining merge module dependencies. The deployment utility hard-coded dependency information for all merge modules TestStand installed. Additionally, no registry existed for all the merge modules installed on a computer, and the deployment utility could use only the merge modules TestStand installed.

Redistribution Using the NI MSI-based Installer Architecture in TestStand 3.5 or Later

To overcome the issues of using merge modules to redistribute products, NI developed an MSI-based installer architecture to create and redistribute parts using the same binary files to install and uninstall parts and to create distributions. Use the NI MSI-based installer architecture to accomplish the following tasks:

- Provide a consistent experience across NI development systems, such as TestStand, LabVIEW, and LabWindows/CVI.
- Create distributions that consist of one developer part, any number of redistributable NI parts, and support files required to install the parts. The

developer part contains files you add to the distribution by using the deployment utility and MSI-based installer tables that perform actions associated with those files during installation, such as registering ActiveX Automation servers, creating shortcuts, and importing hardware configuration files into Measurement & Automation Explorer.

- Upgrade the NI MSI-based installer architecture independently of upgrading TestStand.
- Define a framework to register products and dependencies during installation.
- Provide a way for products to define different flavors, which are subsets of features that you can redistribute, such as a run-time version that excludes development features.

Related concepts:

- [Microsoft Windows Merge Modules](#)
- [NI MSI-based Installer Architecture](#)

Using Custom Commands to Execute Third-Party Installers

You can use the TestStand Deployment Utility with custom commands to build a single MSI-based installer that installs all the components of a test system, such as sequences, step types, code modules, support files, user interfaces, NI drivers, run-time engines, and third-party drivers.

Use the Custom Commands dialog box to configure commands the MSI-based installer executes after all files have been installed, such as launching an installer you built with LabVIEW or LabWindows/CVI, a third-party installer, or an installer for a user interface.

Requiring users to sequentially run multiple installers for third-party or separate components of the test system forces users to wait for each installer to complete before running the next installer. This process can introduce errors if users do not run the installers in the correct order.

Use custom commands to accomplish the following tasks:

- Run a single installer to install the entire test system and any required third-party installers
- Serialize the execution of multiple installers in the order you configure in the

deployment utility

- Request a reboot after all commands execute
- Display a progress window to aid in debugging installation issues
- Use macros in command-line arguments to pass TestStand-specific directories to a third-party installer. For example, the <TESTSTAND> macro corresponds to the <TestStand> directory for the version of TestStand you used to build the installer.

Related concepts:

- [TestStand Directory Structure](#)

Clearing the Product Cache

The product cache, located at <Program Files>\National Instruments\Shared\ProductCache and at <ProgramData>\National Instruments\MDF\ProductCache on Microsoft Windows 10/8.1/7, contains copies of NI installers you can include in the installer you build with the TestStand Deployment Utility.

If the deployment utility cannot locate the required installers in the product cache, the utility prompts you to copy the installers from the appropriate NI distribution media before you create the deployment. Enable the **Copy The Selected Installers And All Future Installers to This Computer** option in the Drivers and Components dialog box for the deployment utility to copy the installers to the product cache when the deployment utility builds an installer that contains drivers and to make all NI products copy the installers to the product cache when you run the installers.

If the deployment utility returns errors when you try to build an installer that includes NI drivers or components, deleting the files in the product cache might resolve the error if the product cache includes out-of-date or corrupt files.

If you do not plan to include NI drivers or components in the installer you build with the deployment utility, you can safely delete the files in the product cache.



Note If you delete files from the product cache, you must exit and restart the deployment utility to refresh the list of files in the cache.

Managing Versioned and Non-Versioned Files

Versioned files are binary files that include a version resource, such as executables, DLLs, or LabVIEW packed project libraries. Non-versioned files are text files, such as `.ini` files, or binary files that do not include a version resource, such as VIs.

MSI-based Installers use the version resource of a file to determine what action to perform when the file already exists on the computer, as shown in the following table.

Version Resource of MSI-based Installer File	Version Resource of File on Computer	Action MSI-based Installer Performs
Later	Earlier	Overwrites file on computer
Earlier	Later	Does not install the file
Any	None	Overwrites file on computer
None	Any	Does not install the file
None	None	Uses creation and modified dates

If you update the version of a file, you do not have to use the Force File to Install option on the Distributed Files tab of the TestStand Deployment Utility to include the file in the deployment.

MSI-based Installers evaluate the creation and modification dates of non-versioned files to determine what action to perform when the file already exists on the computer, as shown in the following table. Right-click a file in Microsoft Windows Explorer and select **Properties** from the context menu to view the modified and created dates.

Modified Date of File on Computer	Created Date of File on Computer	Action MSI-based Installer Performs
Later than created date	—	Does not install the file (prevents overwriting changes made to the file after original installation)
—	Later than modified date	Overwrites file on computer
Same as created date	Same as modified date	Overwrites file on computer

Related concepts:

- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)

Upgrade Codes That Affect MSI-based Installer Behavior

MSI-based Installer upgrade codes uniquely identify MSI-based installers. MSI-based Installers that use the same upgrade code are assumed to install the same set of files or product, such as two different versions of the same test system. MSI-based Installers with different upgrade codes are considered independent and can be installed on one computer at the same time.

MSI-based Installers with the same upgrade code use the installer version to determine what action to perform, as shown in the following table.

Version of New MSI-based Installer	Version of MSI-based Installer on Computer	Action MSI-based Installer Performs
Later	Earlier	Installs and upgrades the previous version of the MSI-based installer on the computer
Earlier	Later	Does not install

MSI-based Installers with Different Upgrade Codes That Install the Same File

Do not include shared files that install to the same path in MSI-based installers that use different upgrade codes because using one of the MSI-based installers to uninstall files removes the shared file, which can negatively affect an installed test system.

To manage shared files correctly with Microsoft Windows Installer technology, the Component ID (GUID) must be the same in each MSI-based installer that shares the file so the MSI Engine can correctly increment the reference count for the shared file and not uninstall the shared file until you uninstall all installers that require the shared file. When multiple installers include shared files with different component IDs, one uninstaller can remove the file from the test station computer, regardless of how many other existing installed test systems on the test station computer require the file. When you use the TestStand Deployment Utility to build MSI-based installers from a single TestStand deployment specification file (.tsd), the MSI-based installers use the same

stored component ID for shared files when the destination paths of the files do not change.

If you are using LabVIEW, you can use packed project libraries to work around this issue because for packed project libraries, the MSI-based installer you build with the deployment utility specifies the `msidbComponentAttributesSharedDllRefCount` attribute, which uses an alternate reference counting mechanism that uses the installation path instead of the component ID. Therefore, MSI-based installers you build with the deployment utility that install a packed project library to the same path increment the same reference count for the file. The packed project libraries uninstall only after you uninstall all MSI-based installers that reference the files.

Related concepts:

- [Choosing Single or Multiple Deployments](#)
- [Relationship of NI MSI-based Installer Architecture and Microsoft Windows Installer Technology](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)

Building a Customized Package Distribution

Use the TestStand Deployment Utility or NI Package Builder to build a package distribution for the deployable image or patch deployable image.

You can customize the package attributes, the dependency relationships, and the inclusion of individual files in the package distribution. If you choose the Repository or Package Installer build outputs, you can include the TestStand Runtime and installers for NI hardware drivers and other software components. If you choose the single package build output, the generated package will require these components to be present on the test station computer, but will not include them in the distribution.

A package installer you build automatically requests a reboot when any of the NI drivers or software components you redistribute require a reboot, such as the TestStand Runtime.



Note To create multiple top-level packages from a single TestStand project,

NI recommends using a supported version of NI Package Builder. Refer to ***Building a Distribution with NI Package Builder*** and the ***NI Package Builder Manual*** for more information.

Upgrading Previous Package Distributions

NI packages are uniquely identified by the package name that you specify. If you create a new package with the same name as another package, NI Package Manager will replace the previous package with the newer package. If the version number of the newer package is less than the existing package, NI Package Manager will indicate that the package is a downgrade.

When you create a patch for a package distribution, the package name must remain the same to ensure that the previous package version is replaced with the updated package.

Troubleshooting Installers You Build with the TestStand Deployment Utility

Use the following techniques to troubleshoot building installers with the TestStand Deployment Utility:

- Review the Status Log on the Build Status tab of the deployment utility for more information about installer build errors. Right-click the Status Log and select **Group By»Severity** or use the column headers to filter and display only errors and warnings to help you locate possible problems with the installer. Click the **Save Log** drop down button and select **Save Technical Support Report** to save a copy of the log that NI can use the log files the deployment utility generates to help you diagnose installer-related issues.
- Ensure the hard drive has sufficient disk space for building the installer. For example, including the TestStand Runtime requires about 450 MB of disk space.

Troubleshooting Installers You Build with the NI Package Builder

NI package builder provides build output and error log panes to help troubleshoot issues with deployment. Refer to NI Package Builder Help for more information on how the application displays error logging and output messages.

Installing Package Distributions Silently

You can install a package distribution without viewing any installation dialog boxes. For single packages, refer to the NI Package Manager Manual for information on silent installation through the command line.

Related concepts:

- [Building a Package Distribution with NI Package Builder](#)

Related information:

- [NI Package Builder Manual](#)
- [NI Package Manager Manual](#)

Patching Deployments

Use the TestStand Deployment Utility and follow the same process NI recommends for creating a full deployment to create or patch deployments and to build installers for patch deployments from an existing TestStand 2013 or later deployment specification file (`.tsd`).



Note

- You cannot create a patch deployment from a TestStand 2012 or earlier `.tsd` file because the file does not include information required to create a patch. You can recreate a new full deployment in TestStand 2013 or later based on the TestStand 2012 or earlier `.tsd` file. You can then create patch deployments from the TestStand 2013 version of the `.tsd` file.
- Once you create the full deployment, you must save the `.tsd` file and the installer for the deployment to create a patch deployment and installer.

For MSI-based installer deployments, you can use patch deployments to update individual files, to add features and add components to features, to return files to a

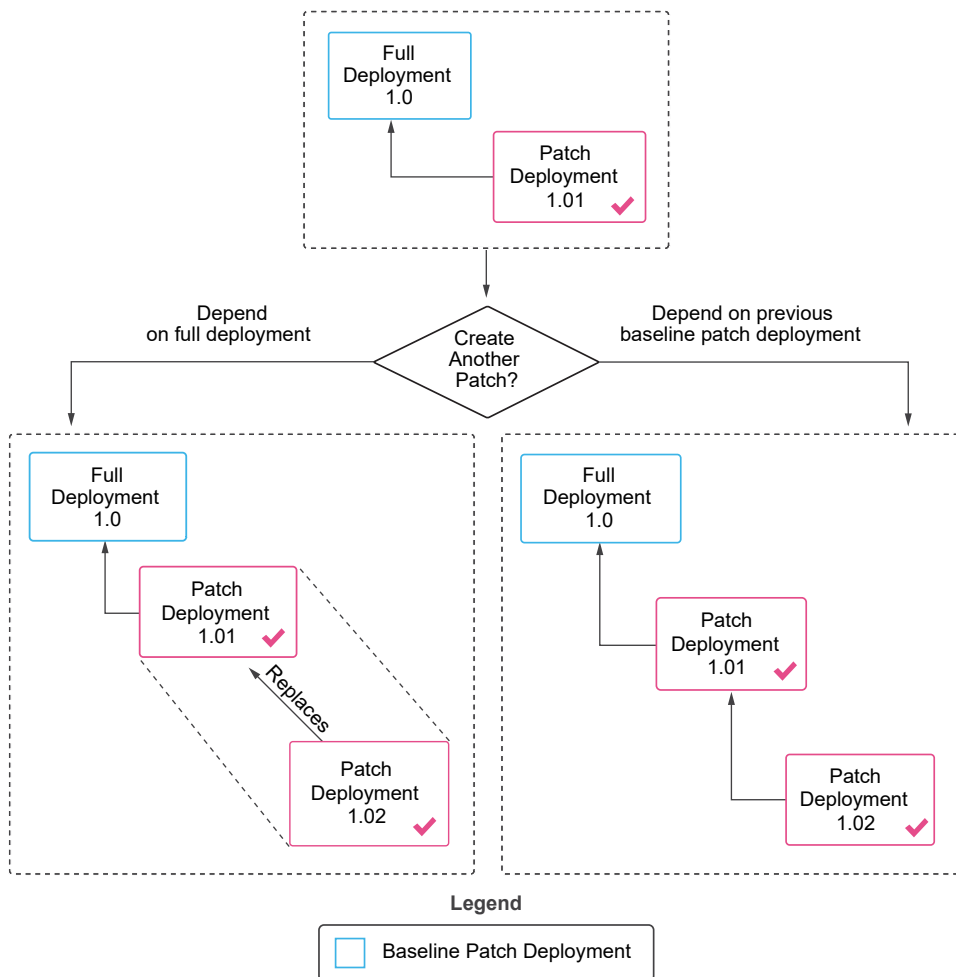
known state, and to include or exclude updates to NI drivers or the TestStand Runtime in a deployed test system. You can also exclude new or modified files from a patch deployment, but you cannot remove files from a deployed system on a test station computer. Patch deployments typically are smaller in size than full deployments and might take less time to create on the development computer.

For package based deployments, you can use patches to include newly required drivers and components, and include updates to components in the deployment. Package based deployments do not support updating individual files through a patch deployment.

You can create a full deployment at any point during the patch deployment creation process. When you change to full deployment mode, options in the deployment utility might change as well. In most cases, the full deployment will be larger and take longer to download and install than a patch deployment.

Creating Cumulative Patch Deployments

By default, the deployment utility creates a cumulative patch deployment that depends on the previous full deployment and includes all the files that differ from the full deployment version of the file. If you make changes to the test system after you create and install a patch deployment on a test station computer, you can create a new patch deployment to include all the changes since the previous full deployment, including the changes in the first patch deployment you created. The newer installer for the patch deployment uses the same upgrade code but a later version number than the previous installer for the patch deployment to upgrade, or replace, the previous patch deployment on the test station computer. You can also create to control the installation order of patch deployments and to include only files modified since a previous patch deployment. The following figure illustrates the process of creating replacement or dependency patch deployments that install serially.



Creating Dependency Patch Deployments from Baseline Patch Deployments

You can create a new, dependency patch deployment from a previous, baseline patch deployment instead of from a full deployment to enforce the order in which to install patch deployments and to minimize the size of a new patch deployment by including only the files that differ from the previous baseline patch deployment version of the file, as shown in the previous figure. When you build an installer for a dependency patch deployment, the deployment utility adds installer launch conditions to enforce installing patch deployments in the correct order.

As the number of patch deployments you create grows, consider creating a new full deployment or a baseline patch deployment.

You must designate a patch deployment as a baseline patch deployment before you create a dependency patch deployment by clicking the **Make Baseline Patch**

Deployment button on the Mode tab of the deployment utility. To create a dependency patch deployment, enable the **Depend on previous Baseline Patch Deployment** option on the Mode tab of the TestStand Deployment Utility.

Because the deployment utility discards old data after you create a deployment, TestStand supports only one baseline patch deployment at a time, and you can create a baseline patch deployment only from the most recent patch deployment you created. If you create another baseline patch deployment, you can no longer depend on the previous baseline patch deployment. Creating a new deployment also erases data of previous patch deployments that you do not designate as baseline patch deployments.

Related concepts:

- [Deployment Process Overview](#)
- [Including Modified Files in a Patch Deployment](#)
- [Adding New Files to a Patch Deployment](#)
- [Including Unmodified Files in a Patch Deployment](#)
- [Including NI Drivers or the TestStand Runtime in a Patch Deployment and Installer](#)
- [Excluding New or Modified Files from a Patch Deployment](#)
- [Upgrade Codes That Affect MSI-based Installer Behavior](#)
- [Creating Deployments](#)
- [Differences between Patch Deployments and Full Deployments](#)
- [Differences between Standard Mode and Manual Mode](#)
- [Creating Independent Patch Deployments](#)
- [Limitations of Patch Deployments](#)

Differences between Patch Deployments and Full Deployments

Use patch deployments to add new or update changed files, drivers, or components in a previously installed full deployment. Patch deployments typically include only the files and installers you modified since creating the full deployment. Avoid using patch deployments when you need to reorganize files or add or modify a significant number of files on the test station computer.

Use full deployments to install a test system to a test station computer that does not include any previously installed version of the deployment. A full deployment includes all the required files and installers you configured in the TestStand deployment

specification file (.tsd) for the deployment. Avoid using full deployments when you need to make only small changes or add or change only a few files on the test station computer.

Deployment Type	Advantages	Disadvantages	Typical Use Cases	Best Practices
Patch deployment	• Smaller than full deployment it updates (Size is proportional to the size and number of changed files, drivers, components, and installers in the test system.) • Installs faster than full deployment it updates • Requires installed full deployment on test station computer to prevent installing files with missing dependencies	• Creating patch installers requires the corresponding full deployment installer • Cannot uninstall the patch installer without uninstalling the full deployment installer • Can make only limited changes	• Fixing bugs in full deployment • Adding new files or features to full deployment	• Create cumulative patch deployments that include changes from all previous patch deployments you created from a full deployment • Create baseline patch deployments, from which you can create dependency patch deployments • Save the .tsd files to store information required to create dependency patch deployments • Validate the test system before

Deployment Type	Advantages	Disadvantages	Typical Use Cases	Best Practices
				deploying patch deployments
Full deployment	• Installs to test station computer that does not have any previously deployed test system • Unlimited changes, such as deleting or moving files, when updating a previous full deployment on the test station computer	• Inefficient for making changes to a subset of files in a previously deployed test system • Same or larger size as previous full deployment, even when you modify only one file • Same or larger size installer, which typically takes longer to install than patch deployments because installers for full deployments uninstall and then reinstall all the files in the deployable image	• Providing foundation from which to create patch deployments • Frequently installing test systems to new test station computers (A single full deployment typically takes less effort to install than installing a full deployment and then installing several patches to update the test system.) • Changing a significant number of files, drivers, or components on the test station computer • Making changes beyond the limits of patches, such as deleting files or reorganizing the	• Save the <code>.tsd</code> file to store information required to create patch deployments • Back up <code>.tsd</code> files so you can revert changes if needed • Save the full installers to store information required to build patch installers

Deployment Type	Advantages	Disadvantages	Typical Use Cases	Best Practices
			directory structure of files included in a previously deployed test system Using different upgrade codes to install multiple versions of a test system side-by-side on test station computers	

Related concepts:

- [Limitations of Patch Deployments](#)
- [Patching Deployments](#)
- [Validating Deployments](#)
- [Upgrade Codes That Affect MSI-based Installer Behavior](#)
- [Creating Independent Patch Deployments](#)
- [Uninstalling a TestStand Deployment](#)

Limitations of Patch Deployments

Limitations for MSI-based installer patches

You can use installers for patch deployments to create smaller updates for a deployment, but patch installers cannot make all the changes that installers for full deployments can complete.

Installers you build using the TestStand Deployment Utility for full and patch deployments use Microsoft Windows Installer technology database files to indicate a Major Upgrade or a Minor Upgrade. Installers for full deployments perform Major Upgrades, which uninstall the previous installer for the full deployment and install the

newer version of the installer for the full deployment. Installers for patch deployments perform only Minor Upgrades, which modify an existing installation without uninstalling the previous installation, leading to the following limitations:

- You cannot uninstall the files for a patch deployment without uninstalling the files for the full deployment.
- Installers for patch deployments cannot delete files. Create a new full deployment to upgrade the previous full deployment to remove files in a deployed test system.
- Installers for patch deployments cannot move files from one directory to another. You can change the destination of a file in a patch deployment, but the installer for the patch deployment leaves the file in the original location of the full deployment and installs a file with the same name in the new destination directory. Because the file remains in the original location, TestStand might find the original file in search paths before finding the new version of the file in a different location. If the file is a VI, LabVIEW might cross-link to the file during a mass compile.



Note NI recommends using a full deployment to delete or move files.

Additionally, the deployment utility does not support the following installer features:

- Select Windows Installer technology patching options, such as the following:
 - Windows Installer patch (.msp) file
 - Small Updates
- Creating new program item entries for patch deployments in the standard Microsoft Windows Control Panel facility for adding and removing programs



Note You can, however, use the **Installation Name** option on the Installer Options tab of the deployment utility and the **Version** option in the Advanced Installer Options dialog box to update the program item entry text for the full installer upon which the patch installer depends.

Refer to Microsoft documentation for more information about these installer features.

Limitations for package-based installer patches

The following limitations exist for package-based package deployments:

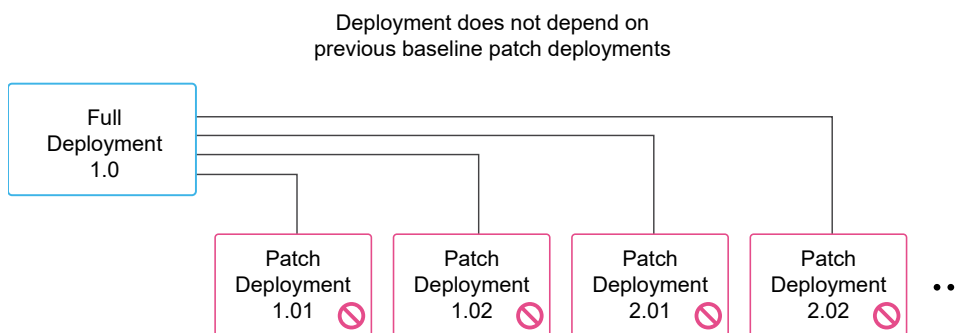
- Patching is not supported for the single package build output. Patching only applies to the driver and component packages included in a repository or package distribution.
- The package company and product fields cannot be modified in a patch distribution since these fields are used to generate the package name. All other package attributes can be modified for a patch distribution.
- Package based deployments do not support updating individual files through a patch deployment. When installing a patch deployment, the package containing deployment files replaces the package installed on the test system.
- Packages for patch deployments cannot move files from one directory to another. You can change the destination of a file in a patch deployment, but the package for the patch deployment leaves the file in the original location of the full deployment and installs a file with the same name in the new destination directory. Because the file remains in the original location, TestStand might find the original file in search paths before finding the new version of the file in a different location. If the file is a VI, LabVIEW might cross-link to the file during a mass compile.

Related concepts:

- [Relationship of NI MSI-based Installer Architecture and Microsoft Windows Installer Technology](#)
- [Uninstalling a TestStand Deployment](#)

Creating Independent Patch Deployments

The TestStand Deployment Utility does not support creating independent patch deployments that install side-by-side, as shown in the following figure.



NI does not support independent patch deployments for the following reasons:

- Independent patch deployments contain only the items you define and do not automatically contain all the updates from previous deployments.
- Installers for independent patch deployments do not include dependencies on other installers for patch deployments, which means that users can determine which independent patch deployments to install and the order in which to install them.
- It can be difficult to maintain test system functionality when independent patch deployments share files with previous patch deployments because for each shared file, the independent patch deployment must include all the changes made in previous patch deployments to ensure that the test station continues to operate correctly.
- Using independent patch deployments can create significant testing burdens because for each independent patch deployment, you must test each permutation of installation order.

NI recommends creating dependency patch deployments to enforce installation order, which can also reduce testing requirements.

Related concepts:

- [Patching Deployments](#)

Including and Excluding Files in MSI-based Installer Patches

Adding New Files to a Patch Deployment

Modify the existing code of a test system to add new projects to a workspace, add new sequence files and any other additional new files to existing TestStand projects, or add new steps to a sequence. When you create a patch deployment, the TestStand Deployment Utility automatically detects the changes and adds new files to the list of top-level files. Use the Distributed Files tab of the deployment utility to change the destination or file properties of the new files.



Note This topic applies to patches you create for MSI-based installer distributions.

Including Modified Files in a Patch Deployment

The TestStand Deployment Utility automatically includes modified files in a patch deployment. When you create a full deployment, the deployment utility stores checksums for each file in the deployment specification file (`.tsd`).

When you create a patch deployment, the deployment utility completes the following tasks:

1. Creates a temporary full deployable image.
2. Compares the new file checksums to the previous file checksums stored in the `.tsd` file to identify modified files.
3. Includes the modified files in the patch deployable image.



Note This topic applies to patches you create for MSI-based installer distributions.

Related concepts:

- [Creating a Deployable Image](#)

Including Unmodified Files in a Patch Deployment

Complete the following steps to include unmodified files in a patch deployment.

1. Select the file in the **Distributed Files** section of the Distributed Files tab of the TestStand Deployment Utility.
2. Select **Included** in the **File Status** control of the File Properties tab.

Selecting the Included option always includes the file in the patch deployment, even if the file is unmodified. Use this technique to return a file that might have been modified on a test station computer to a known state or to set installer properties,

such as registering an ActiveX Automation server or creating a new program item.



Note This topic applies to patches you create for MSI-based installer distributions.

Including NI Drivers or the TestStand Runtime in a Patch Deployment and Installer

Use the Upgrades and Patches to Include option on the Mode tab of the TestStand Deployment Utility to include or exclude updates to NI drivers or the TestStand Runtime included in the full deployment installer in an installer for a patch deployment. To add additional NI drivers or TestStand Runtime components to the patch deployment installer, use the Installer Options tab of the deployment utility and click the **Drivers and Components** button to launch the Drivers and Components dialog box and select the additional components.



Note This topic applies to patches you create for MSI-based installer distributions.

Excluding New or Modified Files from a Patch Deployment

Complete the following steps to exclude new or modified files from a patch deployment.

1. Select the file in the **Distributed Files** section of the Distributed Files tab of the TestStand Deployment Utility.
2. Select **Not Included** in the **File Status** control of the File Properties tab.

Selecting the Not Included option always excludes the file in the patch deployment, even if the file is modified.



Note Excluding files from a patch deployment can negatively affect a deployed test system.



Note This topic applies to patches you create for MSI-based installer distributions.

Differences between Standard Mode and Manual Mode

The default User Files to Include and Upgrades and Patches to Include options on the Mode tab of the TestStand Deployment Utility include built-in safeguards to help you ensure that the test system can execute without errors on the test station computer when you install a patch deployment. NI recommends using the default values for these options in almost all cases to ensure you include all the required files in a patch deployment. The default value for the User Files to Include option is Modified Files. The default value for the Upgrades and Patches to Include option is TestStand Runtime and Drivers.

Select **Manual Mode** in the **User Files to Include** option only when you are certain you know which files to include in or exclude from a patch deployment or installer. When you select Manual Mode, you can use the techniques in the following table to complete certain tasks:

Technique	Deployment Utility Behavior	When to Use	Additional Actions to Complete
Create patch deployments without analyzing the files during the build process, which can reduce the amount of time the deployment utility requires to build an installer for a patch deployment.	Uses the previous analysis results for the deployment.	When you know exactly which files you want to include.	Click the Analyze Source Files button on the Distributed Files tab to begin analyzing all files included in the patch deployment.

Technique	Deployment Utility Behavior	When to Use	Additional Actions to Complete
Use the Add Patch File button on the Distributed Files tab to add arbitrary files to the patch deployment.	Adds these files without processing them.	When you want to add a file without finding the dependencies.	To process the files you add, disable the Include without Processing Item or Dependencies option on the File Properties tab of the Distributed Files tab and click the Analyze Source Files button to analyze the files you add.
Select a file to replace an original file in the deployable image.	Does not automatically include dependencies for the replacement file.	When you have an updated version of a file located in the non-default location that you want to include in the patch deployment.	—

When you use Manual Mode to create a patch deployment, the deployment utility generates additional data for files you add to and replace in the deployment. The deployment utility discards the additional data for the files you add after you create a full deployment or a patch deployment using Standard Mode. The deployment utility maintains the additional data for replacement files until you delete it.

Example

Complete the following steps to use the deployment utility to include a single modified file in a patch deployment.

1. Load the TestStand deployment specification file (.tsd) you used to create the full deployment.
2. Select **Create new patch deployment** on the Mode tab.
3. Select **Manual Mode** in the **User Files to Include** option.
4. Click the **Distributed Files** tab, select **View Build Preview**, and select the modified file you want to add to the patch deployment.
5. Click **Build** to create the patch deployment that includes only the modified file you selected.



Note This topic applies to patches you create for MSI-based installer distributions.

ADE-specific Issues for MSI-based Installer Patches

Patching LabVIEW VIs

When you create a patch deployment for a test system that uses LabVIEW VIs, consider the following issues:

- Removing files from an LLB, packed project library, project library, or a project and creating a patch deployment might negatively affect sequence files or VIs in the existing deployed test system. To ensure compatibility, only add—do not delete—files to an LLB, packed project library, project library or project.
- If you do remove files from an LLB, verify that previous deployments do not require the files. Do not manually remove unchanged code module VIs from the source files. Instead, rely on the TestStand Deployment Utility to remove files that do not differ from the previous deployable image.
- The deployment utility automatically processes VIs by updating links to subVIs and other dependencies to ensure the VIs can execute without errors when run using the LabVIEW Run-Time Engine on a test station computer without the LabVIEW development system installed. Do not use a patch deployment to replace a VI with an unprocessed VI because doing so can break the VI because of missing dependencies or broken links to dependencies. Instead, make the changes to the source VIs and rely on the deployment utility to process the files and include required files in the patch deployable image.



Note This topic applies to patches you create for MSI-based installer distributions.

Related concepts:

- [Patching Deployments](#)
- [Organizing Test Program Files with LabVIEW Libraries](#)
- [Updating and Patching VIs in LabVIEW Projects](#)
- [Updating and Patching VIs in LabVIEW Packed Project Libraries](#)

- [Updating and Patching Stand-alone VIs](#)

Patching LabWindows/CVI DLLs and Executables

When you create a patch deployment for a test system that uses LabWindows/CVI DLLs or executables, consider the following issues:

- You must add new DLL or executable dependencies, such as adding a new `.uir` file for a new dialog box, to the TestStand workspace to ensure the TestStand Deployment Utility includes the files in the patch deployment.
- If you build a DLL or executable with a newer version of LabWindows/CVI, you might need to install the new version of the LabWindows/CVI Run-Time Engine on the test station computer.
- Update the file version when you update a DLL or executable so installers can use the file version number to determine whether to install the DLL or executable on the test station computer.



Note This topic applies to patches you create for MSI-based installer distributions.

Related concepts:

- [Patching Deployments](#)
- [Managing Versioned and Non-Versioned Files](#)

Patching Microsoft .NET Assemblies

When you create a patch deployment for a test system that uses Microsoft .NET assemblies, consider the following issues:

- You must manually add new dependent assemblies to the patch deployment image to ensure the TestStand Deployment Utility includes the files in the patch deployment.
- If you build a DLL or executable against a newer version of .NET, you must install

the new version of .NET on the test station computer.

- Update the file version when you update an assembly so installers can use the file version number to determine whether to install the assembly on the test station computer.



Note This topic applies to patches you create for MSI-based installer distributions.

Related concepts:

- [Patching Deployments](#)
- [Managing Versioned and Non-Versioned Files](#)

Debugging a Patch Deployment

Creating a patch deployment might require more care than creating a full deployment because the changes you include in a patch deployment might negatively affect files in the deployed test system on the test station computer. To avoid this situation, ensure that you correctly prepare the source components for the patch deployment and that you analyze the test system before you create the deployment.

The TestStand Deployment Utility includes several errors and warnings specifically related to patch deployments.

Use the Build Status tab of the deployment utility to view information about the patch deployment, such as which files the patch deployment modifies or adds. The Patch Files tab of the Build Summary message in the Status Log includes specific information about all the files detected, whether the utility included the files, and the checksums used to compare against the previous versions of the files. If the Status Log indicates that the patch deployment does not include a file you expect it to, select **View Build Preview** in the **View** control on the Distributed Files tab of the deployment utility to verify that the correct source file is included. In addition, verify that the source file contains all changes made since the previous deployment.

Related concepts:

- [Preparing Source Components for Deployment](#)
- [Analyzing the Test System](#)

Patching Custom Sequence Editors

If you deployed a test system you created with a custom sequence editor, depending on the type of changes you make, you might need to complete the following tasks:

- Create a patch deployment for the custom sequence editor itself on the development computer on which the custom sequence editor resides and install the custom sequence editor patch on test station computers and any other development computers that require the custom sequence editor.
- Create a patch deployment for the updated test system you created with the custom sequence editor on the development computer on which the custom sequence editor resides and install the patch deployment on test station computers you want to update.

Reverting a Patch Deployment

Use the following table to determine how to revert a patch deployment based on the deployment mechanism you use.

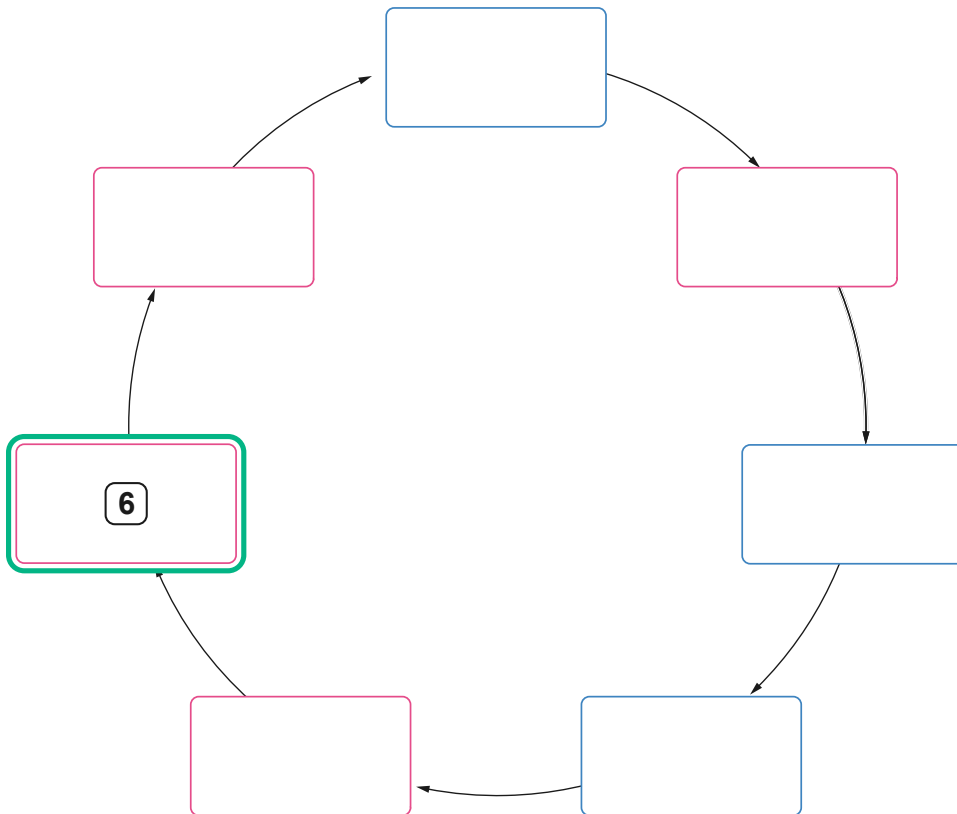
Table 44. Reverting a Patch Deployment

Deployment Mechanism	Action to Take to Revert Patch Deployment
Installers	1. Uninstall the files for the full deployment, which uninstalls all associated patch deployments. 2. Reinstall the version of the full deployment and previous patch deployments you want.
Source Code Control System	Force-synchronize files to a date earlier than the patch deployment you want to revert.
Network Drive	1. Create a list of all files in the patch deployable image. 2. For each file, determine if the file exists in the full deployable image or in previous patch deployable images. 3. If the file exists in the deployable image, copy the latest version of the file to the network drive. If the file does not exist in the deployable image, delete the file from the network drive.

Related concepts:

- [Choosing a Deployment Mechanism](#)
- [Building a Customized MSI-based Installer](#)
- [Using a Source Code Control System to Deploy Test Systems](#)
- [Using a Network Drive to Deploy Test Systems](#)

Transferring Files



You can transfer files from a development computer to a test station computer by using distribution media, by copying files from a local network, or by using email or FTP, depending on the location of the test station computers and available resources.

Consider the following factors when you select a transfer method:

- **Capacity**—Determine how large a file you need to transfer. For example, a standard CD holds approximately 650 MB, a single-layer DVD holds approximately 4700 MB, and a single-layer Blu-Ray holds approximately 25 GB. You can use the TestStand Deployment Utility to build an installer that spans multiple discs, but this technique requires an operator to manually change the discs during the

installation process. Internet connections might not have strict limits, but you might incur additional charges to transfer large amounts of data through email or FTP.

- **Speed**—Determine the time required to transfer files. Consider the total time required to physically move the distribution media to the test station computer and read the information from the media to transfer the entire test system, which can directly affect manufacturing downtime.

Related concepts:

- [Network-Based Deployment Mechanisms](#)
- [Building a Customized MSI-based Installer](#)

Deploying Multiple Test Systems to the Same Test Station

Multiple test systems can exist on the same test station computer. For example, you might want a legacy version and the current version of a test system on the same test station computer. When you install deployments for different versions of TestStand on the same computer, use the TestStand Version Selector to select the active version of TestStand before you launch the deployed TestStand application. NI does not support running applications that use two different versions of TestStand at the same time.

To effectively use multiple test systems on the same test station computer, you must design the test systems to avoid potential conflicts when the test systems or . If you are creating MSI-based installers, use the Save As option in the TestStand Deployment Utility and select the option to create a new installer to create a new TSD file with a different installer GUID so both installers can install side by side. You can also create installers that install side by side.

If you are creating package distributions, ensure that the product and/or company name fields are different for each distribution, since these fields are used to generate the package name which must be unique for each package you create.

The following changes you make to one test system affect all test systems installed on a test station computer:



Note You must apply bug fixes and updates for common files to all test

system locations if each test system uses a separate copy of the common files.

- Changing TestStand configuration settings stored in `GeneralEngine.cfg`, `Adapters.cfg`, `TypePallette.cfg`, and `SearchDirectories.cfg` affects all test systems on the test station computer. For example, if one test system adds a directory to the search paths, another test system on the test station computer can also find a file in that directory. You can use LabVIEW projects, unique filenames in each test system, or different configuration directories for each test system on the test station computer to work around this behavior. If you want to use a separate configuration file for multiple deployments on a single test station, you can deploy the files using an engine environment file to specify a custom location for the configuration files for each deployment. See the Deploying with Environments topic for more information.
- Updating NI drivers affects all test systems on the test station computer.

Supporting Multiple Versions of TestStand

When you use the deployment utility to build an installer, the deployment utility can include multiple versions of the TestStand Runtime. The Include TestStand Runtime option includes only the version of the TestStand Runtime that matches the version of the deployment utility. Use the Drivers and Components dialog box to include additional TestStand Runtimes. Multiple versions of the TestStand Runtime can exist on the test station computer, but the test system uses the most recently registered version of TestStand. You can use the TestStand Version Selector to change the currently active version of TestStand.

Test Systems That Do not Share Files or Execute at the Same Time

Test systems that do not share files and that do not load or execute simultaneously rarely conflict with each other when you deploy the test systems to the same test station computer.

To ensure that the test systems do not share files, install the test systems to different directory hierarchies on the test station computer and ensure that each test system uses separate copies of the code modules, including code modules used in all test systems on the test station computer. If the test systems do not share files, you can

update any file in one test system without affecting another test system.

To ensure that the files do not load in memory simultaneously, unload all code modules after you execute one test system and before you load the next test system to avoid conflicts among the test systems installed on the same test station computer.

Test Systems That Share Files

Shared files must work correctly with all the test systems that use the shared files. You must test changes to shared files with all the test systems on the test station computer to ensure that the test systems execute successfully after the update.

Consider the following guidelines when you change shared files:

- If you change the prototypes of exposed interfaces of shared files, you must update all the callers in all the test systems that use the shared files to ensure that the callers execute without errors. Exposed interfaces include VI connector panes and DLL function prototypes.
- Do not delete files until you confirm that no test system installed on the test station computer uses the files.
- Do not install an updated version of the file unless both test systems can use the updated version of the file.
- Compiling a test system to use a later version of LabVIEW also updates any shared files to use the later version of LabVIEW, which can break VIs in the test systems you did not update.
- Uninstalling an installer you built with the TestStand Deployment Utility removes shared files from other test systems installed on the test station computer unless you distributed the files as LabVIEW packed project libraries.
- When you update existing test systems, the installer replaces a previously installed configuration file or does not modify the file. When you replace a configuration file, such as `GeneralEngine.cfg`, you replace all the settings stored in the file. TestStand does not merge the existing settings with the new settings in the file. Because configuration files are non-versioned files, you can use the Force File to Install option on the Distributed Files tab of the deployment utility to ensure that deployments always install configuration files, such as `GeneralEngine.cfg`. If you want to use a separate configuration file for multiple deployments on a single test station, you can deploy the files using an engine environment file to specify a custom location for the configuration files for each deployment. See the Deploying

with Environments topic for more information.



Notice When you enable the Force File to Install option, an installer can replace a newer version of a file with an older version of a file, which can be problematic for DLLs because components on the computer might rely on functionality available only in the newer version of the DLL and might not execute correctly when the installer downgrades the DLL.

Test Systems That Execute Simultaneously

Name collisions or type conflicts can occur if you load sequence files from multiple test systems simultaneously. Name collisions can occur among VIs or among dependent DLLs. Type conflicts can occur when TestStand loads two sequence files that include data types or step types with the same type version but different definitions.

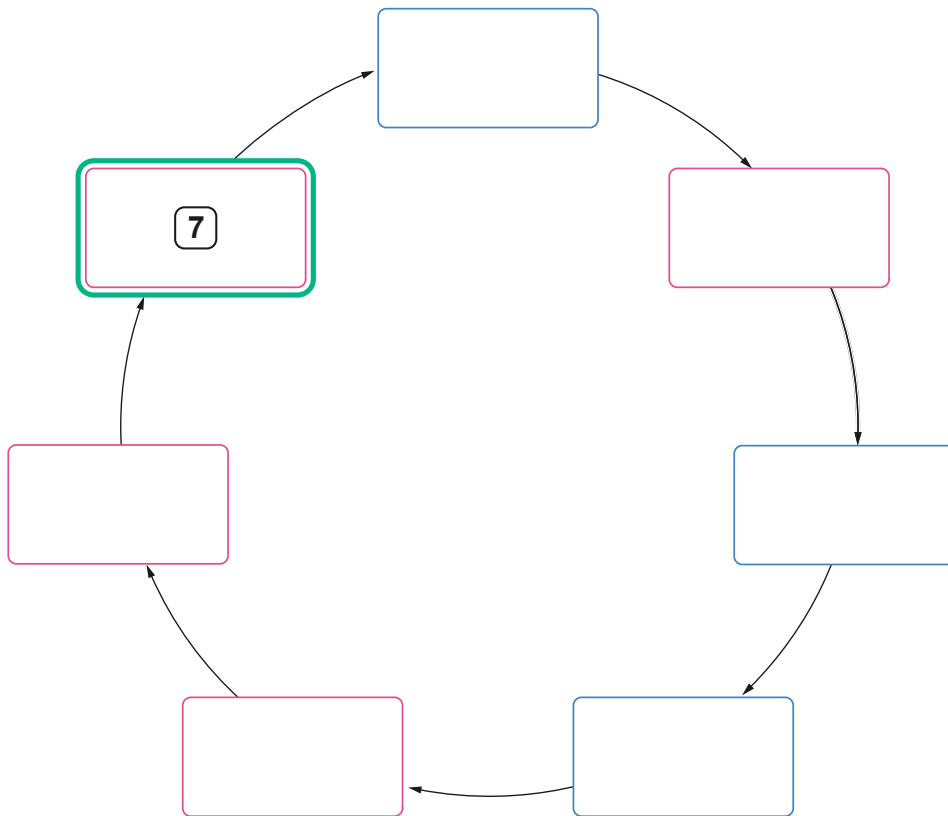
Use the following techniques to avoid VI and DLL dependency name collisions or type conflicts:

- Use unique filenames or unique qualified names for VIs across test systems you install on the same test station computer.
- Call VIs in the context of LabVIEW projects that are unique to each test system to avoid name collisions and partial project library conflicts.
- Because TestStand cannot load two DLLs that load dependencies with the same name from different paths, change the DLLs to use the same dependency with the same path or use unique names for the dependencies.
- Download the NI TestStand Toolkit for Large Application Development from ni.com and use the type differ tool to locate type differences between sequence files of two test systems.

Related concepts:

- [Building a Customized MSI-based Installer](#)
- [Choosing Single or Multiple Deployments](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Managing Versioned and Non-Versioned Files](#)
- [Managing Type Revisions](#)
- [Partial Project Libraries](#)

Validating Deployments



Before you deploy a test system to production computers, complete the following steps.

1. Test the deployment on the development computer you used to create the deployment.
2. Test the deployable image.
3. Test the installer or package distribution you build with the TestStand Deployment Utility.
4. Validate the test system and user interfaces on test station computers.

Testing on Development Computers

Development computers include the necessary tools to fix issues in the test system and rebuild the deployment, but you cannot use development computers to fully test whether the installer you build with the deployment utility installs all the required dependencies because the development computer already includes these files. For example, if the test system requires a DLL in the `Windows\System` directory or a .NET assembly in the Global Assembly Cache (GAC) and you do not include those

components in the deployment, the test system most likely functions correctly on the development computer because the files already exist on the computer. However, when you deploy the test system to another computer that does not already include those files, the test system might not function correctly because the deployment did not include those files.

Testing Deployable Images

When you specify the files to include in the deployment on the System Source tab of the deployment utility, the deployment utility copies all the files and their statically referenced dependencies to a directory called the deployable image. The deployment utility creates the deployable image regardless of the deployment mechanism you use.

Use the following techniques to test the deployable image and resolve issues you discover before you create an installer with the deployment utility:

- Run the TestStand Sequence Analyzer on all the sequence files in the deployable image directory to confirm that the deployment includes all the necessary dependencies. This technique works only when the test system does not use absolute paths because the analyzer uses the source file from the absolute path instead of the copy of the file in the deployable image directory.
- Execute the test system from the deployable image directory. This technique works only when the development computer includes the hardware the test system requires and when the test system does not use absolute paths because the test system executes source files from the absolute paths and does not execute the copies of the files in the deployable image directory. You might execute a combination of source files and files from the deployable image directory if the deployment includes files from the <TestStand>, <TestStand Public>, or <TestStand Application Data> directories or if you are using search directories. To ensure that TestStand loads the correct file, copy the files from the deployable image directory to the <TestStand>, <TestStand Public>, <TestStand Application Data>, or to the correct search directory on the development computer. You also might need to back up the deployable image directory or perform TestStand configuration changes before you execute the test system to avoid modifying or adding files to the deployable image directory during testing. For example, executing a test system might create or modify files in the deployable image directory or log test results to manufacturing databases.

Testing Distributions You Build with the Deployment Utility

Do not test an installer you build with the deployment utility on the development computer for the following reasons:

- If the installer you build with the deployment utility installs files to the same locations as the source files on the development computer, the installer overwrites those original files, which might cause you to lose work. The installer provides no warning that it overwrites the source files.
- If the installer you build with the deployment utility contains a driver with an install type other than `Full`, installing the driver might remove features of the driver from the development computer.
- Testing a deployment installer on the development computer does not detect missing dependent files, components, or drivers because the development computer already includes these files.

To effectively test the installer or package you build with the deployment utility, install the deployment on a separate computer that has only the operating system installed. If you are testing a single package or repository distribution type, you will need to install NI Package Manager on the test machine before installing the package.

Testing on Test Stations

Using a test station computer to test the deployment can help you identify additional required components or dependencies. When practical, test deployments on all supported operating systems on an extra, inactive test station computer that is identical to production test station computers to prevent disrupting the manufacturing process. If you do not have an extra test station computer, configure a virtual machine with the supported operating systems to test the deployment for missing code modules or required drivers. If the test system uses a hardware abstraction layer, run validation tests on the virtual machine using a software simulation of the hardware to identify issues with the deployment.

You can also run the TestStand Sequence Analyzer on the test station computer to identify issues to correct.

Validation Testing

Write validation tests for basic functionality of the test system to verify that the test system performs as designed. Validation tests use known inputs for the test system and verify that the test system returns the correct results. For example, you can test a functioning UUT and a UUT with known defects to confirm that the test system reports the outcome of testing the UUTs correctly. Many test systems use simulated measurements for testing to manage the difficulty of maintaining a large inventory of UUTs for validation.

User Interface Testing

Test the functionality of user interfaces as you develop them and validate that the test station computers include the dependencies the user interfaces require, such as files the user interface loads or references dynamically. Testing user interfaces on test station computers can also help you identify differences between development computers and test station computers, such as screen resolution issues.

Previous:Transferring Files

Related concepts:

- [Choosing a Deployment Mechanism](#)
- [TestStand Directory Structure](#)
- [Identifying Components to Deploy](#)
- [Transferring Files](#)

Common Deployment Scenarios

To complete the following examples that describe how to use the TestStand Deployment Utility in common deployment scenarios, you need one development computer that contains a complete installation of TestStand and one test station computer.

- Deploying the TestStand Runtime
- Distributing Tests from a Workspace
- Adding Dynamically Called Files to a Workspace
- Distributing a User Interface

To run the TestStand Sequence Editor or User Interface application on the test station computer, you must activate an appropriate license or run the application in Evaluation Mode.

Related concepts:

- [Deploying the TestStand Runtime](#)
- [Distributing Tests from a Workspace](#)
- [Adding Dynamically Called Files to a Workspace](#)
- [Distributing a User Interface](#)
- [Activating and Licensing TestStand](#)

Deploying the TestStand Runtime

Complete the following steps to deploy the TestStand Runtime.

1. Select **Tools»Deploy TestStand System** in the TestStand Sequence Editor to launch the TestStand Deployment Utility.
2. On the System Source tab, enable the **Deploy Files»From TestStand Public Directories** option to collect files from the <TestStand Public> directories. When you select this option, the deployment utility distributes to the test station computer any component file customizations, such as process models, step types, user interfaces, and language strings, you saved in the <TestStand Public> subdirectories.

Option A: MSI-based Installer

Complete the following steps to configure the MSI-based installer.

1. On the Mode tab, select the **MSI-based installer** output type.
2. On the Installer Options tab, use the **Installer Directory** control to specify the directory to which the deployment utility copies the installer for the test system.
3. On the Installer Options tab, enable the **Install TestStand Runtime** option.
4. Click the **Drivers and Components** button to launch the Drivers and Components dialog box, in which you can select additional components to include in the installer.
5. To include development components such as the TestStand sequence editor, disable the **Only display runtime installers** option to display the TestStand

product.

6. Select the **TestStand** product to include the sequence editor in the TestStand Runtime installation.
7. Click **OK** to apply the new settings and close the dialog box.
8. Click **Save** and save the build as `EngineInstaller.tsd`.
9. Click the **Build** button to build the installer.

Option B: Package-based distribution

Complete the following steps to configure the package distribution:

1. On the Mode tab, select the **Package-based Distribution** output type, then select the **Package Installer** option.
2. On the Package Distribution Options tab, use the **Output Directory** control to specify the directory to which the deployment utility copies the installer for the test system.
3. On the Package Distribution Options tab, enable the **Install TestStand Runtime** option.
4. Click the **Dependencies** button to launch the Drivers and Components dialog box, in which you can select additional components to include in the distribution.
5. Select the **TestStand Sequence Editor** product to include the sequence editor in the TestStand Runtime installation.
6. Click **OK** to apply the new settings and close the dialog box.
7. Click **Save** to save the build as `EnginePackageInstaller.tsd`.
8. Click **Build** to build the package installer.

Installing the Test System

Complete the following steps to install the test system.

1. To use the installer, copy all the files from the installer directory you specified on the Installer Options tab to the distribution media you want to use or to a shared directory on a network.
2. On the test station computer, insert the distribution media or connect to the network and run the installer executable to start the installer.
3. When the installation completes, complete the following steps to verify that the TestStand Runtime installed correctly.
 - Launch NI Package Manager

- Select the Installed tab, and enter teststand in the search field.
 - Verify that the TestStand products you included are present in the list of packages.
4. Activate a license when the sequence editor prompts you.

Related concepts:

- [TestStand Directory Structure](#)
- [Activating and Licensing TestStand](#)
- [Distributing a User Interface](#)

Distributing Tests from a Workspace

Complete the following steps to distribute tests from a workspace.

1. Select **Tools»Deploy TestStand System** in the TestStand Sequence Editor to launch the TestStand Deployment Utility.
2. On the System Source tab, enable the **Deploy Files»From TestStand Workspace File** option.
3. Click the **File Browse** button located to the right of the From TestStand Workspace File control.
4. Browse to the `<TestStand Public>\Examples\Fundamentals\Deployment` directory and select the `test.tsw` workspace file.
5. Click **Open**.
6. Click the Distributed Files tab.
7. Click **Yes** in the dialog box that launches to allow the deployment utility to analyze the workspace file and dependent files.
8. Click the **Build Status** tab and review the Status Log to check for analysis result warnings.

Configuring the Installer

Complete the following steps to configure the installer.

1. On the Installer Options tab, use the **Installer Directory** control to specify the directory to which the deployment utility copies the installer for the test system.
2. Disable the **unused.dll** option to remove it from the deployment because the example test system does not use this DLL.

3. On the Installer Options tab, enable the **Install TestStand Runtime** option.
4. Click the **Drivers and Components** button to launch the Drivers and Components dialog box, in which you can select additional components to include in the installer.
5. Disable the **Only display runtime installers** option to display the TestStand product.
6. Select the **TestStand** product to include the sequence editor in the TestStand Runtime installation.
7. Click **OK** to apply the new settings and close the dialog box.
8. Click **Save** to save the build as `test.tsd`.
9. Click the **Build** button to build the installer.

Package-based distribution

Complete the following steps to configure the package distribution:

1. On the Mode tab, select the **Package-based Distribution** output type, then select the **Package Installer** option.
2. On the Package Distribution Options tab, use the **Output Directory** control to specify the directory to which the deployment utility copies the installer for the test system.
3. On the Package Distribution Options tab, enable the **Install TestStand Runtime** option.
4. Click the **Dependencies** button to launch the Drivers and Components dialog box, in which you can select additional components to include in the distribution.
5. Select the **TestStand Sequence Editor** product to include the sequence editor in the TestStand Runtime installation.
6. Click **OK** to apply the new settings and close the dialog box.
7. Click **Save** to save the build as `EnginePackageInstaller.tsd`.
8. Click **Build** to build the package installer.

Installing the Test System

Complete the following steps to install the test system.

1. To use the installer, copy all the files from the installer directory you specified on the Installer Options tab to the distribution media you want to use or to a shared directory on a network.

2. On the test station computer, insert the distribution media or connect to the network and run the installer executable to start the installer.
3. When the installation completes, complete the following steps to verify that the TestStand Runtime installed correctly.
 - Launch NI Package Manager
 - Select the Installed tab, and enter teststand in the search field.
 - Verify that the TestStand products you included are present in the list of packages.
4. Activate a license when the sequence editor prompts you.
5. Verify the installation by loading and running `<TestStand Public>\Examples\Fundamentals\Deployment\test.seq`.

Related concepts:

- [TestStand Directory Structure](#)
- [Activating and Licensing TestStand](#)
- [Adding Dynamically Called Files to a Workspace](#)
- [Distributing a User Interface](#)

Adding Dynamically Called Files to a Workspace

Complete the following steps to add dynamically called files to a workspace.

1. Select **Tools»Deploy TestStand System** in the TestStand Sequence Editor to launch the TestStand Deployment Utility.
2. On the System Source tab, enable the **Deploy Files»From TestStand Workspace File** option. To deploy a dynamically called file, you must add the file to a workspace that you deploy.
3. Click the **File Browse** button located to the right of the From TestStand Workspace File control.
4. Browse to the `<TestStand Public>\Examples\Fundamentals\Deployment` directory and select the `Dynamically_called_sequence.tsw` workspace file.
5. Click **Open**.
6. Click the Distributed Files tab.
7. Click **Yes** in the dialog box that launches to allow the deployment utility to analyze the workspace file and dependent files.

8. Click the **Build Status** tab and review the Status Log, which reports a warning that an expression calls a sequence file you might need to add to the workspace.
9. Click the **Distributed Files** tab and notice that `Dynamic.seq` is missing in the Distributed Files list.
10. In the sequence editor, open the `<TestStand Public>\Examples\Fundamentals\Deployment\Dynamically_called_sequence.tsw` workspace file.
11. Add `<TestStand Public>\Examples\Fundamentals\Deployment\Dynamic.seq` to the workspace file and save the changes.
12. On the Distributed Files tab of the deployment utility, click the **Analyze Source Files** button to analyze the modified workspace file. Notice that the Distributed File list now includes `Dynamic.seq`.
13. Click the Build Status tab and review the Status Log, which reports a warning about an expression calling a sequence file. You can ignore the warning because you just added the correct sequence to the workspace.

Configuring the Installer

Complete the following steps to configure the installer.

1. On the Installer Options tab, use the **Installer Directory** control to specify the directory to which the deployment utility copies the installer for the test system and enable the **Install TestStand Runtime** option.
2. Click the **Drivers and Components** button to launch the Drivers and Components dialog box, in which you can select additional components to include in the installer.
3. Disable the **Only display runtime installers** option to display the TestStand product.
4. Select the **TestStand** product to include the sequence editor in the TestStand Runtime installation.
5. Click **OK** to apply the new settings and close the dialog box.
6. Click **Save** to save the build as `Dynamic.tsd`.
7. Click the **Build** button to build the installer.

Package-based distribution

Complete the following steps to configure the package distribution:

1. On the Mode tab, select the **Package-based Distribution** output type, then select the **Package Installer** option.
2. On the Package Distribution Options tab, use the **Output Directory** control to specify the directory to which the deployment utility copies the installer for the test system.
3. On the Package Distribution Options tab, enable the **Install TestStand Runtime** option.
4. Click the **Dependencies** button to launch the Drivers and Components dialog box, in which you can select additional components to include in the distribution.
5. Select the **TestStand Sequence Editor** product to include the sequence editor in the TestStand Runtime installation.
6. Click **OK** to apply the new settings and close the dialog box.
7. Click **Save** to save the build as EnginePackageInstaller.tsd.
8. Click **Build** to build the package installer.

Installing the Test System

Complete the following steps to install the test system.

1. To use the installer, copy all the files from the installer directory you specified on the Installer Options tab to the distribution media you want to use or to a shared directory on a network.
2. On the test station computer, insert the distribution media or connect to the network and run the installer executable to start the installer.
3. When the installation completes, complete the following steps to verify that the TestStand Runtime installed correctly.
 - Launch NI Package Manager
 - Select the Installed tab, and enter teststand in the search field.
 - Verify that the TestStand products you included are present in the list of packages.
4. Activate a license when the sequence editor prompts you.
5. Verify the installation by loading and running `<TestStand Public>\Examples\Fundamentals\Deployment\Call_sequence_dynamically.seq`.
6. On the Mode tab, select the **MSI-based installer** output type.

Related concepts:

- [TestStand Directory Structure](#)
- [Activating and Licensing TestStand](#)
- [Distributing a User Interface](#)
- [Distributing Tests from a Workspace](#)

Distributing a User Interface



Note You must install LabVIEW or LabWindows/CVI Run-Time Engine version and the .NET version that corresponds to the development environment version you use to create user interfaces on the test station computer.

Complete the following steps to distribute a user interface.

1. In the TestStand Sequence Editor, select **File»New»Workspace File** to create and save a new workspace file as `Deploy User Interface.tsw`.
2. Right-click the Workspace window and select **Insert New Project into Workspace** from the context menu.
3. Save the project as `User Interface.tpj`.
4. Right-click the **User Interface** project and select **Add Files to Project** from the context menu.
5. In the file browse dialog box, browse to the `<TestStand Public>\UserInterfaces\Simple\CVI\SourceCode` directory and change the **Files of Type** setting to **All Files (*.*)**.
6. Select `TestExec.exe` and `TestExec.uir` and click **Add**. If TestStand prompts you to resolve the path, select the **Use a relative path for the file you selected** option and enable the **Apply to All** option.
7. Click **OK** to close the dialog boxes.
8. Save the workspace file.

Distributing Tests from a Workspace

Complete the following steps to distribute tests from a workspace.

1. Select **Tools»Deploy TestStand System** in the sequence editor to launch the TestStand Deployment Utility.
2. On the System Source tab, enable the **Deploy Files»From TestStand Workspace File** option.

3. Click the **File Browse** button located to the right of the From TestStand Workspace File control.
4. Browse to the workspace file you saved in .
5. Click **Open**.
6. Click the Distributed Files tab.
7. Click **Yes** in the dialog box that launches to allow the deployment utility to analyze the workspace file and dependent files.
8. Select `TestExec.exe` in the Distributed Files list. The File Properties section to the right of the Distributed Files list updates for this selection.
9. In the File Properties section on the Distributed Files tab, enable the **Create Program Item** option.
10. Enter `Simple CVI UI` in the associated string control to add a shortcut menu item for `TestExec.exe`.

Option A: Configuring the Installer

Complete the following steps to configure the installer.

1. On the Mode tab, select the **MSI-based installer** output type.
2. On the Installer Options tab, use the **Installer Directory** control to specify the directory to which the deployment utility copies the installer for the test system and enable the **Install TestStand Runtime** option.
3. Click **Save** to save the build as `SimpleCVIUI.tsd`.
4. Click the **Build** button to build the installer.

Option B: Configuring the Package Distribution

Complete the following steps to configure the installer:

1. On the Mode tab, select the **Package-based distribution** output type, then select the **Package Installer** option.
2. On the Package Distribution Options tab, use the **Output Directory** control to specify the directory to which the deployment utility copies the installer for the test system and enable the **Install TestStand Runtime** option.
3. Click **Save** and save the build as `SimpleCVIUI.tsd`.
4. Click **Build** to build the installer.

Installing the Test System

Complete the following steps to install the test system.

1. To use the installer, copy all the files from the installer directory you specified on the Installer Options tab to the distribution media you want to use or to a shared directory on a network.
2. On the test station computer, insert the distribution media or connect to the network and run the installer executable to start the installer.
3. When the installation completes, complete the following steps to verify that the TestStand Runtime installed correctly.
 - Launch NI Package Manager
 - Select the Installed tab, and enter teststand in the search field.
 - Verify that the TestStand products you included are present in the list of packages.
4. Activate a license when the sequence editor prompts you.

Related concepts:

- [Using Side-by-Side Versions of the LabWindows/CVI Run-Time Engine with TestStand](#)
- [TestStand Directory Structure](#)
- [Activating and Licensing TestStand](#)

Improving Performance for Creating Deployments

Consider the following guidelines for reducing the time required to create a deployment:

- Divide the test system into multiple, smaller deployments to avoid rebuilding parts of the test system that have not changed. For example, if you include the TestStand Runtime and NI driver components in separate deployments, you do not have to redeploy those components until you install a new version of TestStand or a new version of the driver.
- Use the **Components to Include** option in the Drivers and Components dialog box to exclude drivers and components the test system does not use, such as unused LabVIEW Run-Time Engines.

- Enable the **To minimize media prompts while building your installers, copy the selected installers and all future installers to this computer** option in the Drivers and Components dialog box to copy all necessary components from a deployment to a permanent location on the local computer before the deployment utility creates the deployment and to copy all NI deployable installers to a permanent location on the local computer when you run the installers.
- Ensure that you mass compile all the VIs in the version of LabVIEW the test system requires before you create a deployment. When you create a deployment, the TestStand Deployment Utility invokes LabVIEW to save copies of all VIs the test system uses. When saving a copy of a VI, LabVIEW automatically recompiles the VI in the following situations:
 - The compiled code of the VI is out of date
 - The compiled code of the VI is saved in a different version of LabVIEW
 - The VI is a source-only VI, which is a VI with separate compiled code, and has not been compiled on the development computer

Because compiling VIs can require a significant amount of time, NI recommends that you compile all the VIs at once instead of recompiling the same set of VIs each time you create a deployment.

- Ensure that you disable the **Check for Broken VIs** options in the LabVIEW VI Options dialog box to prevent the deployment utility from checking for broken VIs before and after creating the deployable image.
- Use packed project libraries because the TestStand Deployment Utility can include a packed project library in the deployment without having to recompile any of the VIs the packed project library contains.
- If you call VIs in the context of multiple LabVIEW projects, enable the **Consolidate Files Shared By Projects** option in the LabVIEW VI Options dialog box to deploy the VIs in a single, merged source distribution when possible instead of using a source distribution for each LabVIEW project. This consolidation increases build speed and decreases disk usage for multiple projects that share common subVIs by making a single source distribution for every project. This optimization is not always possible because two projects might compile the same source VI differently, which can result in slower build times.
- If you are deploying VIs using a version of LabVIEW released after the current TestStand version, creating the first deployment takes longer because the deployment utility compiles all the files it requires in the newer version of LabVIEW. Creating subsequent deployments does not take as long.

Related concepts:

- [Choosing Single or Multiple Deployments](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Organizing LabVIEW-Based TestStand Systems](#)

Automating Deployment with TestStand Deployment Utility

Use the `BuildTSD.exe` executable to automate builds of TestStand Deployment Utility files and to make modifications to deployment utility files before the build occurs. `BuildTSD.exe` accepts a `.tsd` deployment configuration file and supports optional parameters for configuring properties of the build. It will also return a numeric code to provide information about whether the build completed successfully or failed due to an error.

Supported Command Line Arguments

`BuildTSD.exe` supports the `<Optional Changes>` with the following command line arguments:

- Mode Tab
- System Source Tab
- Installer Options Tab
- Advanced Options Dialog
- Drivers and Components Dialog
- Process Options
- Patching Specific Options

Mode Tab

Option	Type	Notes
<code>/outputMode</code>	Enumeration	Possible values: • Image • MSIInstaller

Option	Type	Notes
		PackageDistribution
/packageDistMode	Enumeration	Possible values: - SinglePackage - Repository - PackageInstaller

System Source Tab

Option	Type	Notes
/deployTSWorkspace	Boolean	
/workspacePath	Absolute Path	You must set the /deployTSWorkspace flag to include the specified workspace in the deployment.
/deployDirectory	Boolean	
/directoryToDeploy	Absolute Path	
/includeSubdirectories	Boolean	
/deployTSPublicDirectories	Boolean	
/deployTSCfgDirectory	Boolean	
/deployableImage	Boolean	
/ActiveEnvironmentFile	Absolute Path	
/DeployEnvironmentFile	Boolean	

Installer Options Tab

Option	Type	Notes
/createInstaller	Boolean	This flag is included for backwards compatibility. For new applications, use the /outputMode flag with the value

Option	Type	Notes
		MSIInstaller instead of this flag.
/installationName	String	
/installerDirectory	String	
/startMenuItemGroup	String	
/defaultInstallationBaseDir	Enumeration	Possible values: • WindowsVolume • Teststand • Windows • System • System32 • ProgramFiles • StartMenu • Startup • Desktop • Personal • Temp • NationalInstruments • TSPublic • TSAppData • AbsolutePath
/doNotAskUserForInstallationDirectory	Boolean	
/defaultInstallationSubdirectory	String	
/installerLanguage	Enumeration	Possible values: • English • French • German • Japanese • Korean
/installTSEngine	Boolean	
/hideLicenseDialog	Boolean	

Advanced Options Dialog

Option	Type	Notes
/manufacturer	String	
/versionString	String	Use format # . # . #
/autoIncrementVersion	Boolean	
/upgradeCode	String	Use a GUID of the form { 21EC2020-3AEA-1069-A2DD-08002B30309D }
/readmeFile	Path	
/licenseAgreement	Path	
/mediaSizeMB	Integer	
/includeNetworkLicenseServerSettings	Boolean	
/specifyConfigurationDirectory	Boolean	
/configDestination	Enumeration	Possible values: • Installation • Teststand • Windows • System • System32 • ProgramFiles • StartMenu • Startup • Desktop • Personal • Temp • NationalInstruments • TSPublic • TSAppData
/configSubDirectory	Relative Path	
/alternateAssociationSeqFiles	Absolute Path	
/processModel	Enumeration	Possible values:

Option	Type	Notes
		• NoChange • ForceDefaultModels • ForceLegacyModels
/installerExecutableName	String	If no value is specified, Deployment Utility uses the default value of <code>install.exe</code> .

Drivers and Components Dialog

Option	Type	Notes
/includeHardwareConfiguration	Boolean	
/hardwareConfigurationFile	Absolute Path	
/displayDialogDuringImport	Boolean	
/MAXImportOptions	Enumeration	Possible values: • Merge • Replace • Preserve

Process Options

Option	Type	Notes
/copyLogFiles <FilePath>	Absolute Path	Copies the log files from the build to the <FilePath> location. The path must be a directory.
/LoadTSDOnly	-	This option will just launch the TSDU and perform the operations specified without automatically performing the build and shutting down. This is useful for checking your command line parameters manually.
/SaveTSDFile <FilePath>	Absolute Path	Saves the <code>.tsd</code> file after the build to the <FilePath> location.



Note The `/LoadTSDOnly` argument should be used only without any extra arguments.

Patching Specific Options

Option	Type	Notes
<code>/SetDistribution</code>	Enumeration	Possible values: • Full • Patch
<code>/UserFilesToInclude</code>	Enumeration	Possible values: • IncludeChangedFiles • DoNotIncludeFiles • Manual
<code>/UpgradesandPatchesToInclude</code>	Enumeration	Possible values: • TSEngineandDrivers • TSEngine • Drivers • None
<code>/DependOn</code>	Enumeration	Possible values: • FullDeployment • BaselinePatchDeployment



Note For boolean options use the strings `TRUE` and `FALSE`.

BuildTSD.exe Exit Codes

The Deployment Utility returns one of the following return values after building a TSD:

Exit Codes

Number	Result	Definition
0	Success	The build succeeded without errors or warnings.
1	Load failed	The TSD file failed to load.
2	Aborted	The TSDU was aborted while building, the installer was not built.
3	Fatal error	The TSDU encountered a fatal error while building.
4	Built with errors	The TSDU created an installer but there were some errors while building.
5	Built with warnings	The TSDU created an installer but there were some warnings while building.
6	Could not start the TSDU	The TSDU could not be found.
7	TSDU did not return a value	This can happen if the TSDU crashes before being able to communicate the result to the caller process.
8	No build performed	TSDU exited without performing a build.

Effectively Using LabVIEW with TestStand

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



Note TestStand does not support calling into malleable VIs (`.vim`).

Related concepts:

- [Deploying TestStand Systems](#)

Organizing LabVIEW-Based TestStand Systems

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

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

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

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

**Note**

- NI recommends using a single version of LabVIEW to compile all the VIs for a test system.
- NI does not recommend using LabVIEW-generated .NET assemblies to develop a TestStand system.

Related concepts:

- [Programming with the TestStand API in LabVIEW](#)
- [Organizing Test Program Files with LabVIEW Projects](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Organizing Test Program Files with Stand-alone VIs](#)
- [Organizing Test Program Files with LabVIEW Libraries](#)
- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)
- [Calling LabVIEW VIs](#)
- [Getting Started with TestStand](#)

Organizing Test Program Files with LabVIEW Projects

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

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

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

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

Choosing When to Execute Code Modules Specified in a LabVIEW Project

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

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

Related concepts:

- [Calling LabVIEW VIs](#)
- [Getting Started with TestStand](#)
- [Programming with the TestStand API in LabVIEW](#)

Editing VIs in LabVIEW Projects

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

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



Note TestStand supports automatically deploying and undeploying shared variables defined in LabVIEW projects.

Related concepts:

- [Organizing Test Program Files with LabVIEW Projects](#)
- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Organizing Test Program Files with LabVIEW Libraries](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Deploying Network-Published Shared Variables](#)
- [Deploying VIs in LabVIEW Projects](#)
- [Updating and Patching VIs in LabVIEW Projects](#)

Updating and Patching VIs in LabVIEW Projects

NI recommends using LabVIEW packed project libraries to modularize a TestStand system. You can create a patch deployment for a modularized system more quickly by rebuilding only the packed project libraries you modify.

If you do not use LabVIEW packed project libraries, the TestStand Deployment Utility might take longer to create the deployment because the utility rebuilds the source distributions for all the VIs in the test system. The deployment utility completes this step to avoid run-time errors that might occur when the LabVIEW Run-Time Engine loads VIs that require recompilation to account for inplaceness changes.

You can use the Force File to Install option on the Distributed Files tab of the deployment utility to install files to guarantee new files install over the files that exist on the target computer.



Note The deployment utility automatically enables the Force File to Install option when you create a patch deployment.

Related concepts:

- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Patching Deployments](#)

- [Deploying TestStand Systems](#)
- [Deploying VIs in LabVIEW Projects](#)
- [Editing VIs in LabVIEW Projects](#)
- [Patching LabVIEW VIs](#)

Deploying VIs in LabVIEW Projects

Consider the following recommendations and requirements for deploying VIs when you decide whether to use LabVIEW projects in a TestStand system:

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

Related concepts:

- [Deploying TestStand Systems](#)
- [Organizing Test Program Files with LabVIEW Projects](#)
- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Editing VIs in LabVIEW Projects](#)
- [Processing LabVIEW Code Modules for Deployment](#)
- [Updating and Patching VIs in LabVIEW Projects](#)

Organizing Test Program Files with LabVIEW Packed Project Libraries

LabVIEW packed project libraries are LabVIEW project libraries that package multiple LabVIEW-related files and dependencies into a single non-editable file (.lvlibp) that you can call from other VIs.

In TestStand, you can directly open and call VIs exported from a LabVIEW packed

project library, or you can access the VIs through a LabVIEW project.

Building Packed Project Libraries with LabVIEW

Although the TestStand Deployment Utility can build packed project libraries that contain code modules the test system uses, using LabVIEW to build packed project libraries offers the following advantages:

- You can create multiple packed project libraries and control which VIs you export from each packed project library. You can also group VIs into packed project libraries depending on functionality.
- You can more quickly create a deployment that contains pre-built packed project libraries because the deployment utility does not need to rebuild the packed project libraries.
- You can rebuild existing packed project libraries to include new exported VIs and maintain compatibility of the previously exported VIs.

Refer to the LabVIEW Help for more information about LabVIEW packed project libraries.

Related concepts:

- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Organizing Test Program Files with LabVIEW Projects](#)
- [Deploying TestStand Systems](#)
- [Calling LabVIEW VIs](#)
- [Getting Started with TestStand](#)
- [Programming with the TestStand API in LabVIEW](#)

Editing VIs in LabVIEW Packed Project Libraries

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

- NI recommends creating an architecture with multiple packed project libraries that group similar sets of VIs together so that you can update code more easily.
- You cannot edit VIs inside packed project libraries. You must have the source VIs and the project that contains the build specification for the packed project library

to make a change to the packed project library. To modify a VI stored in a packed project library, you must modify the source VI and rebuild the packed project library, and replace the older packed project library with the newer version.

- NI recommends enabling the **Callers adapt at run time to Exported VI connector pane state** option on the Connector Pane State page of the Packed Library Properties dialog box in the packed project library build specification in LabVIEW when using other LabVIEW VIs to call exported VIs stored in the LabVIEW packed project library. When you enable this option, VI callers of a packed project library can adapt to inplaceness changes in an exported VI without having to recompile.
- NI recommends that you wait until you have finished making major changes to VIs before building the packed project library. If VIs that use the packed project library are in the same project, you can replace all instances of VIs in the project library with a call to a built packed project library in the LabVIEW Project Explorer window.
- You cannot load a VI stored in a packed project library independently of other items in the same packed project library. LabVIEW loads every VI in the packed project library when you call the first VI, resulting in a longer initial load time and higher memory usage. LabVIEW loads subsequent VIs immediately.
- When you call the first VI in a packed project library, LabVIEW consolidates types in all VIs in the packed project library to reduce the time required to load every VI inside the packed project library. This behavior is faster than loading the same VIs if they were stored outside of the packed project library. Packed project libraries are also faster to load because they consist of a single file, so the test system does not have to search the disk for additional files.
- Packed project libraries support debugging when built with debugging information. However, including the debugging information makes the packed project library larger and slightly slower to execute. Consider benchmarking the differences in size and execution speed to determine the impact of including the debugging information in the packed project library.

Related concepts:

- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Organizing Test Program Files with LabVIEW Projects](#)
- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Deploying TestStand Systems](#)
- [Deploying VIs in LabVIEW Packed Project Libraries](#)
- [Updating and Patching VIs in LabVIEW Packed Project Libraries](#)

Updating and Patching VIs in LabVIEW Packed Project Libraries

LabVIEW packed project libraries are designed to make patching easier. Packed project libraries have an MSI-recognized version resource number, which you can use to more easily make partial updates to the test system. You can replace a packed project library with a newer version without changing the VI or the TestStand steps that call into the packed project library.

By default, the TestStand Deployment Utility automatically includes files that differ from the baseline deployment version of the file. You must compile all LabVIEW packed project libraries before you create a patch deployment to ensure that the deployment utility correctly identifies modified libraries to include in the patch deployment.

Related concepts:

- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Managing Versioned and Non-Versioned Files](#)
- [Patching Deployments](#)
- [Deploying TestStand Systems](#)
- [Deploying VIs in LabVIEW Packed Project Libraries](#)
- [Editing VIs in LabVIEW Packed Project Libraries](#)
- [Patching LabVIEW VIs](#)

Deploying VIs in LabVIEW Packed Project Libraries

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

- Packed project libraries reduce the number of files required to create a deployment using the TestStand Deployment Utility and reduce the total build time because packed project libraries are precompiled files.
- Packed project libraries are compressed files and therefore occupy less disk space than VIs.
- You must determine whether the test system will benefit more from the faster build times the pre-built packed project libraries offer or the convenience of using the deployment utility to build the packed project library:

- If you are deploying a pre-built packed project library, the deployment utility copies only the packed project library.
- If you configure the deployment utility to create a packed project library, it must create the packed project library on every build.
- Packed project libraries are compressed files and therefore occupy less disk space than VIs.
- The deployment utility supports building the packed project library with debugging information to support debugging of the packed project library on the deployed system.



Note

- Use the Deploy Library step type to deploy or undeploy to the local computer the shared variables defined in a LabVIEW project library file or packed project library file, or to deploy or undeploy shared variables defined in LabVIEW project libraries within a selected LabVIEW project.
- The project library or packed project library must contain only shared variables and cannot contain any VI files.
- TestStand cannot auto-deploy shared variables defined in a LabVIEW packed project library specified in a LabVIEW project or in a LabVIEW project library saved in an LLB specified in a LabVIEW project.

Related concepts:

- [Deploying TestStand Systems](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Deploying Network-Published Shared Variables](#)
- [Creating a Deployable Image](#)
- [Editing VIs in LabVIEW Packed Project Libraries](#)
- [Processing LabVIEW Code Modules for Deployment](#)
- [Updating and Patching VIs in LabVIEW Packed Project Libraries](#)

Organizing Test Program Files with Stand-alone VIs

A stand-alone VI is a file on disk that is not part of a LabVIEW library and is not called in the context of a project.

Code module VIs use functions that manipulate input from the user interface or other sources and display that information or move it to other files or computers.

Related concepts:

- [Organizing Test Program Files with LabVIEW Libraries](#)
- [Organizing Test Program Files with LabVIEW Projects](#)
- [Calling LabVIEW VIs](#)
- [Getting Started with TestStand](#)
- [Programming with the TestStand API in LabVIEW](#)

Editing Stand-alone VIs

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

- NI recommends that you save all code modules in one version of LabVIEW. VIs saved in different versions of LabVIEW cannot share data when executed using the LabVIEW Run-Time Engine (RTE).
- Loading multiple stand-alone VIs can negatively impact performance, especially if the test sequences load VIs dynamically.
- Using multiple stand-alone VIs can result in name conflicts. LabVIEW does not support loading multiple VIs with the same name from different locations. Use LabVIEW project libraries and packed project libraries to add a namespace to the VIs and avoid name conflicts.
- NI does not recommend editing packaged VIs on a deployed system because unexpected errors, such as broken VIs or run-time errors, can occur. NI recommends rebuilding and redeploying the deployment image in this situation. Analysis and instrument drivers are examples of VIs that use project libraries.

Related concepts:

- [Organizing Test Program Files with Stand-alone VIs](#)
- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Deploying TestStand Systems](#)
- [Creating Deployments](#)
- [Deploying Stand-alone VIs](#)

- [Updating and Patching Stand-alone VIs](#)

Updating and Patching Stand-alone VIs

Consider the following recommendations and requirements for updating and patching VIs when you decide whether to use stand-alone VIs in a TestStand system:

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



Note NI recommends using LabVIEW packed project libraries to modularize a TestStand system. If you do not use LabVIEW packed project libraries, the TestStand Deployment Utility might take longer to create the deployment because the utility rebuilds the source distributions for all the VIs in the test system. The deployment utility completes this step to avoid run-time errors that might occur when the LabVIEW Run-Time Engine loads VIs that require recompilation to account for inplaceness changes.

- Stand-alone VIs do not have version resources. If a file does not have a version resource, installers use a heuristic process to determine whether to upgrade the file. These heuristics might cause confusion when VIs are not upgraded as you expected, such as a test system in which multiple installers update a single set of utility VIs. You can use the Force File to Install option on the Distributed Files tab of the deployment utility to install files regardless of the heuristic to guarantee new files install over the files that exist on the target computer.



Note The deployment utility automatically enables the Force File to Install option when you create a patch deployment.

Related concepts:

- [Organizing Test Program Files with Stand-alone VIs](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)

- [Managing Versioned and Non-Versioned Files](#)
- [Deploying Stand-alone VIs](#)
- [Deploying TestStand Systems](#)
- [Editing Stand-alone VIs](#)
- [Patching LabVIEW VIs](#)

Deploying Stand-alone VIs

Consider the following recommendations and requirements for deploying VIs when you decide whether to use stand-alone VIs in a TestStand system:

- NI does not recommend distributing two VIs with the same name to different locations on a target computer. When a VI with the same name from a different location is in memory, LabVIEW uses the VI in memory when attempting to load a subVI. However, LabVIEW reports an error when you attempt to load a top-level VI with the same name as a VI in memory even when the similarly named VI in memory is an exact copy of the one you want to load.
- TestStand does not support including two VIs with the same name in a deployment unless the VIs are in different project libraries.
- When you build a TestStand deployment that calls VIs that use shared variables, you must complete the following tasks:
 - Add project library files that contain shared variables to the workspace you use to create the deployment or deploy the files using a LabVIEW Utility Deploy Library step in the test sequences. The TestStand Deployment Utility automatically includes project library files it finds in the workspace or from LabVIEW Utility Deploy Library step references when you configure the step to deploy the project library.
 - Add an aliases file to the deployment and configure the deployment to install the aliases file to the proper location on the destination system so LabVIEW can resolve network paths.
 - Include the NI Variable Engine component in the installer to ensure that the destination system can use the shared variables.
- You must execute a LabVIEW Utility Deploy Library step in a TestStand sequence to deploy shared variables to the local computer. You can also manually deploy shared variables to the local computer using a project in the LabVIEW development system.

Related concepts:

- [Deploying TestStand Systems](#)
- [Organizing Test Program Files with Stand-alone VIs](#)
- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Deploying Network-Published Shared Variables](#)
- [Resolving References to Shared Variables](#)
- [Editing Stand-alone VIs](#)
- [Processing LabVIEW Code Modules for Deployment](#)
- [Updating and Patching Stand-alone VIs](#)

Organizing Test Program Files with LabVIEW Libraries

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

Consider the following when working with LLBs:

- An installer cannot install a single VI to an LLB. You must overwrite the entire LLB to install additional files.
- When you make changes to an LLB, it is difficult to determine which files have changed.
- LLBs do not have hierarchy information, so all VIs are stored at the same level. However, for display purposes, you can mark VIs called from TestStand as top-level VIs in the LLB.
- LLBs do not have namespaces and cannot prevent name collisions.
- All VIs in an LLB are visible, so you cannot hide the names of private VIs and subVIs.
- LLBs do not have version resources, so multiple installers upgrading a single LLB might have unintended consequences.
- Removing files from an LLB and creating a patch deployment might break sequence files or VIs in the existing deployed test system. If you remove files from an LLB, verify that previous deployments do not require the files. Do not manually remove unchanged code module VIs from the patch deployable image. Instead, rely on the TestStand Deployment Utility to remove files that do not differ from the previous deployable image.

If you want to build LLBs of test VIs when creating a deployment of a test system, you

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

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

Related concepts:

- [Deploying TestStand Systems](#)
- [Patching Deployments](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Patching LabVIEW VIs](#)
- [Processing LabVIEW Code Modules for Deployment](#)

Organizing Test Program Files with LabVIEW Project Libraries

LabVIEW project libraries (`.lvlib`) are collections of VIs, type definitions, shared variables, palette menu files, and other files, including other project libraries.

In TestStand, you can call public VIs in a project library, but you cannot call private VIs. To call a VI in a project library, you must specify the path to the VI file. TestStand does not use qualified paths that include the project file name and VI name.

Using project libraries in TestStand includes the following advantages:

- You can avoid potential VI name collisions when you deploy multiple test systems to the same test station computer. Project libraries change the filenames of VIs they reference to a compound, qualified name that includes the project library filename so you can reference VIs with the same filename from different project libraries. For example, the `Library.lvlib` project library changes the filename of the `test.vi` it references to `Library.lvlib:test.vi`, and the `LibraryTwo.lvlib` project library changes the filename of the `test.vi` it references to `LibraryTwo.lvlib:test.vi`. When you reference a project library that another project library contains, the qualified name concatenates the parent project library name with the name of the project library it contains, such as `ParentLibrary.lvlib:ReferencedLibrary.lvlib`.

- You can enforce security settings, such as limiting public access to certain files or limiting editing permission for a collection of files to prevent calling support VIs the project library references or calling VIs the project library does not reference.
- You can organize files into a single hierarchy of items. When you reference a VI a project library references, LabVIEW also automatically references all the VIs in the project library. Similarly, when you load a VI a project library references, LabVIEW automatically loads the project library as well.

The project library must exist on the computer for TestStand or LabVIEW to execute a VI the project library references. If the project library is not available, LabVIEW breaks the VI to prevent it from running.

Refer to the LabVIEW Help for more information about project libraries.

Related concepts:

- [Deploying Multiple Test Systems to the Same Test Station](#)
- [Deploying TestStand Systems](#)
- [Programming with the TestStand API in LabVIEW](#)

Organizing Test Program Files with LabVIEW-Built Shared Libraries

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

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

Refer to LabVIEW documentation for information about debugging LabVIEW DLLs.



Note

- To ensure that Microsoft Windows messages are handled during the execution of a LabVIEW DLL you build for use with TestStand, NI recommends that you disable the **Delay operating system messages in shared library** option on the Advanced Page of the Shared Library Properties dialog box in the LabVIEW Professional Development System.
- LabVIEW does not support exporting every LabVIEW data type. You might have difficulty passing data to certain LabVIEW types from TestStand. You must also manually add parameter information when configuring the LabVIEW-generated DLL in TestStand.
- When you deploy a test system, you must install the version of the LabVIEW RTE that corresponds to the version of LabVIEW used to create the DLL on any computer that uses the DLL. You can include the appropriate RTE installer in an installer you build with the TestStand Deployment Utility.
- TestStand supports—but NI does not recommend—distributing DLLs with the same name that steps in a sequence file call.

Related concepts:

- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)
- [Organizing Test Program Files with LabVIEW Projects](#)
- [Deploying TestStand Systems](#)
- [Processing LabVIEW Code Modules for Deployment](#)

Special Considerations for Using LabVIEW with TestStand Systems

Learn the recommendations and restrictions for using LabVIEW components with TestStand. You might need to complete additional work to ensure that the LabVIEW and TestStand components interoperate correctly.

Related concepts:

- [Managing Memory Allocation for Large LabVIEW Data Sets](#)

Managing Memory Allocation for Large LabVIEW Data Sets

LabVIEW automatically handles memory allocation. Because this process is automatic, LabVIEW copies data frequently.

If the test system involves large sets of data, large and frequent data copies might result in an out-of-memory error.

Related concepts:

- [Special Considerations for Using LabVIEW with TestStand Systems](#)
- [Reusing VIs](#)

Reusing VIs

If a user interface calls a VI that the test program also calls as a code module, you must ensure that the name of the VI called from the user interface does not match the name of the VI when it is called as a code module.

Apply a prefix to all dependent VIs when building the user interface into a stand-alone application to avoid a naming conflict among the dependencies of the user interface executable and the code module VIs called from test sequences.



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

Related concepts:

- [Using TestStand UI Controls in LabVIEW](#)
- [Deploying Network-Published Shared Variables](#)
- [Managing Memory Allocation for Large LabVIEW Data Sets](#)
- [Reserving Loaded VIs for Execution](#)

Reserving Loaded VIs for Execution

Enable the **Reserve Loaded VIs for Execution** option in the LabVIEW Adapter Configuration dialog box to reserve any VIs TestStand loads for calling with the LabVIEW Adapter.

Use this option to avoid reloading VIs when they are used and to make references you create in a VI you call from TestStand—such as I/O, ActiveX, and synchronization references—persist across calls to other VIs. You can store these references in a TestStand property and pass them to subsequent VIs you call from TestStand.

Editing a Reserved VI



When you open a reserved VI in LabVIEW, the Run arrow indicates the VI is reserved, and you cannot edit the VI.



To edit a VI TestStand has reserved, click the **Edit VI** button on the LabVIEW Module tab in the TestStand Sequence Editor or on the Module tab of the Edit LabVIEW VI Call dialog box in a TestStand User Interface. You can also right-click the step and select **Edit Code** from the context menu to open the VI in TestStand.

You can also select **File»Unload All Modules** in the TestStand Sequence Editor before you open the VI in LabVIEW.

Closing References to a Reserved VI

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

Reserving VIs Affects the LabVIEW VI Server Run VI Method

By default, TestStand enables the **Reserve Loaded VIs for Execution** option in the LabVIEW Adapter Configuration dialog box so that all VIs TestStand loads remain in memory. Enabling this option improves execution time because TestStand does not need to load and unload VIs.

When running a VI reserved for execution, LabVIEW creates a proxy VI to call the VI configured in the TestStand step as a subVI of the proxy. Therefore, LabVIEW considers VIs called from TestStand as subVIs.

When you open and run a subVI as a top-level VI in LabVIEW, LabVIEW returns error code 1000. Additionally, if you use the Open VI Reference function to open a VI that TestStand marks as reserved for execution, the VI opens as a top-level VI, and you then use the VI Server Run VI method, LabVIEW returns error 1000.

You can use the following techniques to work around this LabVIEW behavior:

- Create a wrapper VI that calls the VI as a subVI. You can then use the Open VI Reference function to open the wrapper VI and run it with the Run VI method of the VI Server.
- Make the VI re-entrant.
- On the Run Options panel of the Step Settings pane of the LabVIEW step that calls the VI, change the **Load Option** setting to **Load dynamically** and the **Unload Option** setting to **Unload after step executes**.

Related concepts:

- [Reusing VIs](#)
- [Partially Specifying LabVIEW Clusters](#)

Partially Specifying LabVIEW Clusters

If you have a cluster with default values and you need to modify certain values on the cluster depending on an operation you want to perform, you can use partial clusters.

In TestStand, you can specify whether to pass default values for individual elements of a cluster control that are recommended or optional. This behavior is referred to as partially specifying a cluster, or partial cluster passing.

To specify that you want to apply the default value to an element of a partial cluster, enable the **Default** option in the VI Parameter Table on the LabVIEW Module tab in the TestStand Sequence Editor or on the Module tab of the Edit LabVIEW VI Call dialog box in a TestStand User Interface.

Related concepts:

- [Reserving Loaded VIs for Execution](#)
- [Executing VIs with Separate Compiled Code](#)

Executing VIs with Separate Compiled Code

In LabVIEW 2010 or later, you can separate compiled code from a VI.

When you open a VI with separate compiled code, LabVIEW generates and saves a VI object file (`.viobj`) with the compiled code.

When you separate compiled code from a VI, recompiling the code of the VI does not create an unsaved change. As a result, if you use a source code control system, you do not have to check out all the files in a hierarchy when you edit a VI.

TestStand supports calling these VIs from LabVIEW. During deployment, the TestStand Deployment Utility converts these VIs with separated compiled code to regular VIs so that TestStand can execute these VIs using the LabVIEW Run-Time Engine on the deployed machine.

Related concepts:

- [Deploying TestStand Systems](#)
- [Partially Specifying LabVIEW Clusters](#)
- [Evaluating Conditional Disable Structures and Symbols](#)
- [Processing LabVIEW Code Modules for Deployment](#)

Evaluating Conditional Disable Structures and Symbols

The Conditional Disable Structure contains one or more subdiagrams, or cases. You can specify conditions for the structure based on custom symbols defined in a project.

For all VIs in the main application instance, TestStand evaluates any conditions that use custom symbols when the VI is recompiled. These expressions are not evaluated again during execution.

Related concepts:

- [Executing VIs with Separate Compiled Code](#)
- [Calling VIs That Create ActiveX References](#)

Calling VIs That Create ActiveX References

LabVIEW might seem to hang with 100% processor use when all of the following conditions are true:

- You call a LabVIEW VI that creates ActiveX references, such as calling the ApplicationMgr.GetEngine method directly.
- You configure the VI to use the **same as caller** execution system.
- You call the VI from a sequence running in a single-threaded apartment thread.

Change the **Preferred Execution System** option in LabVIEW to a value different from **same as caller** to work around this issue.

Related concepts:

- [Evaluating Conditional Disable Structures and Symbols](#)
- [Calling VIs That Use XControls](#)

Calling VIs That Use XControls

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

Related concepts:

- [Calling VIs That Create ActiveX References](#)
- [Building a TestStand Deployment with LabVIEW](#)

Building a TestStand Deployment with LabVIEW

Refer to the Deploying TestStand Systems help for the following information:

- Preparing source components for deployment
- Processing LabVIEW code modules for deployment
- Troubleshooting LabVIEW code module issues

- Using multiple LabVIEW versions in a test system

Related concepts:

- [Deploying TestStand Systems](#)
- [Processing LabVIEW Code Modules for Deployment](#)
- [Troubleshooting LabVIEW Code Module Issues](#)
- [Using Multiple LabVIEW Versions in a Test System](#)
- [Calling VIs That Use XControls](#)
- [Creating Custom Step Types that Call VIs for Substeps](#)
- [Deploying Stand-alone VIs](#)
- [Deploying VIs in LabVIEW Packed Project Libraries](#)
- [Deploying VIs in LabVIEW Projects](#)

Creating Custom Step Types that Call VIs for Substeps

When you create a custom step type that defines a Pre-Step, Post-Step, or Edit substep that calls a VI, you must create a source distribution to ensure that TestStand can resolve the file paths to all the dependencies of the VI when TestStand executes the custom step type. If you created the VI using LabVIEW 2010 or later, you can create a LabVIEW packed project library to consolidate all dependencies into a single file that the step type calls.

When you create a source distribution or packed project library, NI recommends that you enable the **Always Run VI in LabVIEW Run-Time** option in the LabVIEW Advanced Settings window in the TestStand Sequence Editor or on the Advanced Settings tab of the Edit LabVIEW VI Call dialog box in a TestStand User Interface to ensure that the VIs execute in the correct version of the LabVIEW Run-Time Engine.

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



Note TestStand supports executing a VI with a LabVIEW Class Reference parameter only when you execute the VI as part of a running sequence. Edit substeps and OnNewStep Custom substeps do not execute as part of sequence execution and cannot contain direct LabVIEW Class Reference

parameters. If you need to pass a LabVIEW Class Reference, serialize the reference to a string parameter.

Related concepts:

- [Creating Custom Step Types](#)
- [Using Substeps](#)
- [Search Directories](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Building a TestStand Deployment with LabVIEW](#)

Calling LabVIEW Class Member VIs from TestStand

Select the **Class Member Call** option from the Call Type ring control on the LabVIEW Module tab in the TestStand Sequence Editor or on the Module tab of the Edit LabVIEW VI Call dialog box in a TestStand User Interface to call a member VI defined in a LabVIEW class, as shown in the following table.

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

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

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



Note

- TestStand does not support dynamic dispatching when you select the **VI Call** option from the Call Type ring control.
- You must have LabVIEW 2012 or later to use LabVIEW dynamic dispatching when calling LabVIEW classes in TestStand.

Related concepts:

- [Passing an Instance of a LabVIEW Class Object](#)
- [Calling LabVIEW VIs](#)
- [Getting Started with TestStand](#)
- [Programming with the TestStand API in LabVIEW](#)

Passing an Instance of a LabVIEW Class Object

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

When TestStand passes an object reference variable that contains a LabVIEW class object to a code module, the class object is always passed by value. To ensure that changes made to a LabVIEW class object persist in a code module, you must pass the class object as an output from the code module and store the output in the TestStand object reference variable. The object reference variable will contain a new instance of the class object with the updated data.

Class Object Lifetime

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

The VI that creates the reference determines the lifetime of LabVIEW reference data

types stored in TestStand. As long as the VI that created the LabVIEW reference is reserved, the LabVIEW reference is valid. This is not the case for LabVIEW class objects. A LabVIEW class object remains valid even after the VI that originally created it is unreserved or closed. If LabVIEW is restarted, trying to use a LabVIEW class object a previous LabVIEW instance obtained results in an error because the LabVIEW class object is not valid in the new instance of LabVIEW. In addition, if you try to use the LabVIEW class object after you close the LabVIEW project context in which the LabVIEW class object exists, an error occurs because closing a LabVIEW project context releases all LabVIEW class objects created in the project context.

Calling a Compatible Class Member VI

When you select the **VI Call** option from the Call Type ring control on the LabVIEW Module tab in the TestStand Sequence Editor or on the Module tab of the Edit LabVIEW VI Call dialog box in a TestStand User Interface, you are responsible for calling the appropriate member VI for the LabVIEW class object stored in TestStand.

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



Note

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

parameter only when you execute the VI as part of a running sequence. Edit substeps and OnNewStep Custom substeps do not execute as part of sequence execution and cannot contain direct LabVIEW Class Reference parameters. If you need to pass a LabVIEW Class Reference, serialize the reference to a string parameter.

Related concepts:

- [Calling LabVIEW Class Member VIs from TestStand](#)
- [Organizing Test Program Files with LabVIEW Projects](#)
- [Calling VIs on Remote Computers](#)

Calling LabVIEW Property Nodes from TestStand

You can configure a TestStand step to call a LabVIEW Property Node, which you can use to configure several properties for LabVIEW I/O references. Complete the following steps to insert a LabVIEW step that configures a LabVIEW Property Node call.

1. Insert an Action step in the Main step group.
2. Select **Property Node Call** from the Call Type ring control LabVIEW Module tab in the TestStand Sequence Editor or on the Module tab of the Edit LabVIEW VI Call dialog box in a TestStand User Interface. The LabVIEW Module tab updates to display the Property Node Type control.
3. Click the **Configure Settings for Property Node**  button located to the right of the Property Node Type control to launch the Configure Property Node dialog box.
4. Select a library type from the Property Node Type control, which displays the different types available in LabVIEW, depending on the hardware configuration.
5. Select the appropriate LabVIEW I/O reference type from the Property Node Class control, which specifies the I/O Reference class for the type you selected in the Property Node Type control. The Available Properties List control lists the properties available for the currently selected LabVIEW I/O reference type or LabVIEW class.
6. Select a property from the Available Properties List control and click the **Add to Selected Properties** button. The selected property appears in the Selected Properties control, and the Description for control updates to display the LabVIEW

description associated with the property. You can also click the **LabVIEW VI Help** button to launch the associated LabVIEW Help topic. Depending on the property, you can change the direction of the property to in or out, or you can use the default (reset) value.

7. Click **OK** to create a VI configured with a Property Node using the specified I/O reference and properties.
8. Close the Configure Property Node dialog box.
9. Save the sequence file.



Note

- You must have LabVIEW 2013 or later to configure and call LabVIEW Property Nodes in TestStand.
- Depending on the hardware configuration on the computer, the Configure Property Node dialog box might list several different Property Node types.
- In LabVIEW, you must configure properties for LabVIEW classes to appear in Property Nodes so that you can select the properties in the Configure Property Node dialog box.

Related concepts:

- [Calling LabVIEW VIs](#)
- [Getting Started with TestStand](#)
- [Programming with the TestStand API in LabVIEW](#)

Debugging and Tracing of VIs Executed Using the LabVIEW Runtime

Using the LabVIEW Development System, you can debug VIs that execute using the LabVIEW Runtime in the TestStand Sequence Editor. Using the LabVIEW Desktop Execution Trace Toolkit (DETT), you can trace and capture events for VIs executed using the LabVIEW Runtime in the TestStand Sequence Editor and the TestStand User Interfaces on the local machine.

The LabVIEW Development System that debugs a VI code module must match the version and bitness of the LabVIEW Runtime that executed the VI code module. In LabVIEW, you can configure breakpoints and probes on diagrams to debug VIs that TestStand executes using the LabVIEW Runtime.

Using the DETT, you can capture and trace events, such as the order of VI execution, thread IDs, reference leaks, timing details, and user-defined trace information. Trace and event information can be helpful when diagnosing complex threading issues, identifying performance bottlenecks, and debugging other complex problems.

To enable debugging and tracing in TestStand, use the **Enable Debugging and Tracing** option in the LabVIEW Adapter Configuration dialog box.



Note

- Changes to the **Enable Debugging and Tracing** setting will not take effect until you restart a TestStand application.
- LabVIEW 2015 or later is required to debug and trace VIs executed in the LabVIEW Runtime.
- DETT 2015 or later is required to trace VIs executed in the LabVIEW Runtime.

Related concepts:

- [Debugging Code Module VIs with LabVIEW](#)
- [Tracing with the DETT](#)

Debugging Code Module VIs with LabVIEW

When executing your VIs in the LabVIEW Development System, you can debug your VIs using LabVIEW by setting breakpoints and probes in VIs being executed. The LabVIEW Development System can also debug applications that execute VIs using the LabVIEW Runtime, such as VIs executed using the TestStand LabVIEW Adapter and LabVIEW-built DLLs executed using the TestStand C/C++ DLL Adapter. Refer to the LabVIEW Help for more information about debugging remote applications and VIs executed using the LabVIEW Runtime.

Complete the following steps to debug VIs when the TestStand Sequence Editor executes VIs using a LabVIEW Runtime:

1. Ensure that the **Enable Debugging and Tracing** option in the LabVIEW Adapter Configuration dialog box is enabled, and restart the TestStand Sequence Editor if required.
2. Ensure that you save all VIs you want to debug in the version of the LabVIEW development system you are using to debug the VIs.
3. Ensure that the LabVIEW code modules you want to debug are loaded into memory. You can load the modules by executing the sequence once, or by starting execution of the sequence if the modules are set to preload when execution begins.
4. In the LabVIEW Development System, open a debug connection to the TestStand Sequence Editor by selecting **Operate»Debug Application or Shared Library** to open the Debug Application or Shared Library dialog box. Set the **Machine name or IP address** to `localhost`, select `seqEdit.exe` in the **Application or shared library** option, and click **Connect**. LabVIEW opens a debug connection to the TestStand Sequence Editor and downloads the code module VIs for debugging. See the Debug Application or Shared Library Dialog Box topic in the LabVIEW Help for more information about this step.
5. In LabVIEW, add the required breakpoints and probes to the downloaded VIs to debug, then save the VI.
6. (Optional) View the VIs loaded for debugging using `LabVIEW RTE Debugging Viewer.vi`, located in `<TestStand Public>/AdapterSupport/LabVIEW`. When you launch this tool, it will display the VIs loaded in the debugging application instance. If no VIs are shown, you can use the **Application to Debug** menu to manually select the application instance you want to view. If none of the options contain `seqEdit.exe`, verify that the debug connection was successful.
7. Execute the TestStand sequence again to execute and debug the VIs. Once the TestStand execution reaches the VI, the first breakpoint will be triggered in a new instance of the VI. To add additional breakpoints or probes, add them to this instance of the VI.
8. To make changes to the VI after debugging, right-click on the front panel of the debugging instance of the VI and select **Remote Debugging»Quit Debug Session**.



Note

- This feature is only supported when executing VIs in the TestStand Sequence Editor.
- Step Into commands in TestStand are not supported when debugging VIs executing using the LabVIEW Runtime.
- You cannot debug TestStand steps configured to execute Express VIs or property nodes.
- LabVIEW will not connect to the TestStand Sequence Editor if the relevant LabVIEW Runtime does not have any VI code modules loaded.

Debugging VIs executing in different LabVIEW projects

If you have a sequence file that executes VIs in one or more project contexts, you can connect to each project context by selecting the item named `SeqEdit.exe - <Project Name>.lvproj - My Computer` from the **Application or shared library** option in the LabVIEW Debug Application or Shared Library dialog box.

Debugging VIs executing in different LabVIEW Runtimes

You can only debug code module VIs executing in one LabVIEW Runtime at a time. If you want to debug VIs executing in a different LabVIEW Runtime, you must close the existing debug connection from LabVIEW and open a new debug connection using the LabVIEW Development System that matches the LabVIEW Runtime version and bitness.

Debugging VIs executed in a User Interface

You can debug VIs executed in a TestStand User Interface by launching the User Interface in the LabVIEW Development System and setting the LabVIEW Adapter to execute VIs using the LabVIEW Development System before you execute your test sequence.

Closing a Debug Connection

When you open a debug connection from LabVIEW, LabVIEW will continue to debug VIs executed by the runtime until all code module VIs being debugged are closed. You can

also close the debug connection using one of the following approaches:

- In the LabVIEW Development System, right-click on the front panel of one of the VIs to debug and select **Remote Debugging»Quit Debug Session** from the shortcut menu.
- In the TestStand Sequence Editor, unload all modules by using either the File menu or the TestStand Engine API.

Tracing with the DETT

The LabVIEW Desktop Execution Trace Toolkit (DETT) is a tool that collects various types of trace data from LabVIEW applications. Refer to the LabVIEW Desktop Execution Trace Toolkit Help and the Opening a Trace Connection topic in the LabVIEW Help for more information about using the DETT.



Note Tracing VI execution using the DETT can have a performance impact on the execution of code module VIs. To minimize the performance impact, connect the DETT as close as possible to the section of code you want to trace.

Configuring the DETT

When you enable debugging and tracing in TestStand, TestStand configures each LabVIEW Runtime loaded by TestStand to open a unique TCP port. Refer to the TCP Ports for the LabVIEW Runtimes topic for more information about TCP ports used.

Complete the following steps to trace and capture events for VIs executed using the LabVIEW Runtime in a TestStand application:

1. Ensure that the **Enable Debugging and Tracing** option in the LabVIEW Adapter Configuration dialog box is enabled and restart the TestStand Sequence Editor or User Interface if required.
2. Execute your test sequence to ensure all LabVIEW Runtimes and VI code modules are loaded.



Note The DETT can only connect to the TestStand Sequence Editor or a

User Interface after TestStand loads a supported LabVIEW Runtime. Determine the appropriate port for the runtime you want to trace. Refer to the TCP Ports for the LabVIEW Runtimes topic for more information.

3. In the DETT:
 - a. Set the **Machine** text box in the toolbar to `localhost`, then set the **Port** text box to the port number for the LabVIEW Runtime you want to trace.
 - b. Refresh the **Application Instance** drop-down menu to see the context for the TestStand application.
 - c. Click the **Start** button in the DETT to begin a trace session.



Note

- You can only trace one LabVIEW execution/project context at a time.
- The DETT can connect and generate trace and event information for the TestStand Sequence Editor and User Interfaces. The DETT cannot connect to a TestStand application until TestStand loads a supported LabVIEW Runtime.

Tracing VIs executing in different LabVIEW projects

If you have a sequence file that executes VIs in one or more project contexts, you must connect to the specific project context you want to trace from the **Application Instance** drop-down menu.

Related concepts:

- [TCP Ports for the LabVIEW Runtimes](#)

TCP Ports for the LabVIEW Runtimes

When the **Enable Debugging and Tracing** LabVIEW Adapter option is enabled, all LabVIEW Runtimes 2015 and later loaded by TestStand open a TCP port. The DETT uses this TCP port to trace VIs executed in the LabVIEW Runtimes.

You can obtain the port number for a specific LabVIEW Runtime by clicking the **TCP**

Ports button in the LabVIEW Adapter Configuration dialog box.



Note TestStand assigns a unique TCP port for each LabVIEW Runtime to avoid Runtimes using the same TCP ports.

Deploying Network-Published Shared Variables

Use network-published shared variables so VIs on distributed systems can share data across the network. LabVIEW identifies shared variables through a network path that includes the computer name (target name), project library name(s), and shared variable name. LabVIEW must create shared variables in the Shared Variable Engine before a test system can access the shared variables.

With a LabVIEW Project

When the test system executes VIs in the context a LabVIEW project, enable the **Auto Deploy Shared Variables on First Load of any VI in a Project** option and the **Auto Undeploy Shared Variables on Last Unload of all VIs in a Project** option in the LabVIEW Adapter Configuration dialog box to automatically deploy and undeploy shared variables defined in the LabVIEW project when the project is first loaded for execution.

Without a LabVIEW Project

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

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



Note

- The project library or packed project library must contain only shared variables and cannot contain any VI files.
- TestStand can deploy a shared variable bound to another shared variable only when you use an NI Publish-Subscribe Protocol (NI-PSP) URL to bind the shared variable to deploy to another shared variable. If you attempt to use a Deploy Library step to deploy a shared variable to a variable in a LabVIEW project, the deployment fails. Refer to the LabVIEW Help for more information about deploying shared variables and NI-PSP URLs.
- NI recommends that you do not use a Deploy Library step to deploy or undeploy shared variables when steps in different threads or executions might access the shared variables.
- TestStand does not support calling a VI or DLL that programmatically deploys shared variables using the LabVIEW Deploy Library method on a LabVIEW Application reference or using the LabVIEW Deploy Items method on a LabVIEW Project reference. When you programmatically deploy shared variables, the TestStand or LabVIEW application might return an error and terminate. You can use the Deploy Library method from within a stand-alone executable without adverse effects on TestStand.
- TestStand cannot auto-deploy shared variables defined in a LabVIEW packed project library specified in a LabVIEW project or in a LabVIEW project library saved in an LLB specified in a LabVIEW project.
- TestStand cannot deploy project libraries that have shared variables that are aliased to existing shared variables in a project or an NI-PSP data item on the network.

Related concepts:

- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Organizing Test Program Files with LabVIEW Projects](#)
- [Organizing Test Program Files with LabVIEW Packed Project Libraries](#)
- [Deploying TestStand Systems](#)

Resolving References to Shared Variables

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

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

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

- **LabVIEW Run-Time Engine**—LabVIEW looks for an aliases file with the same name and location as the TestStand application, such as `SeqEdit.aliases` for the TestStand Sequence Editor and `TestExec.aliases` for a TestStand User Interface. You must quit and restart the TestStand application after you copy the aliases file from the project directory to the application directory.
- **Development System**—LabVIEW looks for an aliases file, `LabVIEW.aliases`, located in the same directory as the LabVIEW development system executable. You must quit and restart LabVIEW after you copy the aliases file from the project directory to the LabVIEW directory.
- **Other Executable**—LabVIEW looks for an aliases file with the same name and location as the LabVIEW executable server, such as `TestStandLVRTS.aliases` for `TestStandLVRTS.exe` and `TestExec.aliases` for a user interface. You must quit and restart the executable after you copy the aliases file from the project directory to the executable directory.

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



Note Ensure the alias file is not read-only to allow LabVIEW to update the

file, if necessary, on the deployed computer.

Related concepts:

- [Organizing Test Program Files with LabVIEW Projects](#)
- [Organizing Test Program Files with LabVIEW Project Libraries](#)
- [Programming with the TestStand API in LabVIEW](#)

Symmetric Multiprocessing in VIs Executed from TestStand

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

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

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



Note

- By default, in scenarios that do not include TestStand, LabVIEW uses multiple threads to execute VIs if the preferred execution system of the VIs is not set to **user interface**.
- LabVIEW always uses a single thread to execute VIs for which the preferred execution system is **user interface**. NI strongly discourages using the **user interface** preferred execution system for VIs you call from TestStand. The remainder of information in this topic excludes

consideration of the **user interface** preferred execution system.

LabVIEW 2009 or Later

The following table shows how these factors apply when you execute VIs using the LabVIEW 2009 or later development system or Run-Time Engine (RTE).

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

LabVIEW 8.6.1 or Earlier

The following shows how these factors apply when you execute VIs using the LabVIEW 8.6.1 or earlier development system or RTE.

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

Configuring CPU Affinity and the Number of LabVIEW RTE Execution Threads

Using the LabVIEW Adapter, you can configure the CPU affinity for LabVIEW development system or RTE execution threads, and you can configure and the number of execution threads the LabVIEW RTE uses to execute VIs. You can adjust these settings to optimize performance for a particular system.

Related concepts:

- [Programming with the TestStand API in LabVIEW](#)
- [Configuring the CPU Affinity for LabVIEW Execution Threads](#)
- [Configuring the Number of LabVIEW RTE Execution Threads](#)
- [Using TestStand on SMP Systems](#)

Configuring the Number of LabVIEW RTE Execution Threads

Using the LabVIEW Adapter, you can configure the number of execution threads the LabVIEW Run-Time Engine (RTE) uses to execute VIs. You can adjust these settings to optimize performance for a particular system.

If the Preferred Execution System option on the Execution page of the VI Properties dialog box in LabVIEW of the VI is **same as caller**, the LabVIEW RTE uses the TestStand execution thread to execute the VI. If you enable the Execute 'Same as caller' VIs Using Multiple Threads option in the LabVIEW Adapter Configuration dialog box, the LabVIEW Adapter configures the LabVIEW RTE to use additional LabVIEW threads, in addition to the TestStand execution thread, to execute the VI.

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

Related concepts:

- [Programming with the TestStand API in LabVIEW](#)
- [Symmetric Multiprocessing in VIs Executed from TestStand](#)

Configuring the CPU Affinity for LabVIEW Execution Threads

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

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

- The CPU affinity of the TestStand execution thread

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

LabVIEW 2009 or Later

The following table shows how these factors apply when TestStand executes VIs using the LabVIEW 2009 or later RTE and the preferred execution system of the VI is **same as caller**.

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

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

LabVIEW 8.6.1 or Earlier

The following table describes the CPU affinity for LabVIEW execution threads when TestStand executes VIs using the LabVIEW 8.6.1 or earlier RTE.

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

General Strategies for Optimizing LabVIEW Code Module Performance on SMP Systems

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

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

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

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

manually tuned the CPU affinity of TestStand execution threads, NI recommends that you run performance tests to determine whether enabling the Additional Threads Inherit Calling Thread's CPU Affinity option in the LabVIEW Adapter Configuration dialog box improves or degrades performance.

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

Related concepts:

- [Symmetric Multiprocessing in VIs Executed from TestStand](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Using TestStand on SMP Systems](#)
- [Configuring the Number of Execution Threads Assigned to a LabVIEW Execution System](#)

Configuring the Number of Execution Threads Assigned to a LabVIEW Execution System

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

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

executable.

For the LabVIEW development system, add the following tokens below the [LabVIEW] tag in the .ini file. For the LabVIEW Run-Time Engine, add the following tokens below the [LVRT] tag in the .ini file.

ESys.StdNParallel = -1

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

where

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

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



Note

- You must restart LabVIEW for the new settings to take effect.
- The settings you specify apply to all versions of the LabVIEW RTE that TestStand loads.

Related concepts:

- [Symmetric Multiprocessing in VIs Executed from TestStand](#)

Calling VIs on Remote Computers

You can directly call VIs on remote computers, including computers that run the LabVIEW development system or a LabVIEW executable and PXI controllers that run the LabVIEW Real-Time Module.



Note TestStand does not support automatically downloading VIs to systems that run the Real-Time Module. To use the Real-Time Module, you must call a VI on the local system and the local VI must then call the VI on the Real-Time Module, or you must manually download the VI to the Real-Time module and call the VI by remote path or by name if the VI is already in memory.

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

Complete the following steps to call a VI remotely:

1. Configure the TestStand step to specify that the call occurs on a remote computer.
2. Configure the remote computer to allow TestStand to call VIs located on the computer.
3. Configure the computer running TestStand to have network access to the remote computer running the LabVIEW VI Server.

Using LabVIEW Class Data Type Parameters in Remote VI Calls

TestStand supports remote VI calls that use LabVIEW class data type parameters only when you preload the LabVIEW class definition in the same version of the LabVIEW Run-Time Engine as the LabVIEW version of the VI on the local computer. Otherwise, LabVIEW fails to interpret the returned LabVIEW class data because it is not loaded into LabVIEW.

Complete the following steps for every step you configure to make a remote VI call that uses LabVIEW class data type parameters to work around this issue.

1. Create another sequence in the same sequence file.

2. Copy all the steps and required variables into the new sequence.
3. For each step in the new sequence, click the **Advanced Settings** button on the LabVIEW Module tab of the Step Settings pane to launch the LabVIEW Advanced Settings window.
4. Disable the **Run VI on a Remote Computer** option and enable the **Always Run VI in LabVIEW Run-Time Engine** option.



Note TestStand preloads the steps in this sequence when it executes the entry point of the current process model.

Calling Class Member VIs on Remote Machines

TestStand does not support remote Class Member calls.

Related concepts:

- [Configuring a LabVIEW Step to Run Remotely](#)
- [Configuring the LabVIEW VI Server to Run VIs Remotely](#)

Configuring a LabVIEW Step to Run Remotely

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

1. Click the **Advanced Settings** button on the LabVIEW Module tab to launch the LabVIEW Advanced Settings window, in which you can specify the name or an expression that evaluates to the name of the remote computer on which you want to run the VI.
2. Specify the path to the remote VI using the following guidelines:

Scenario	Control to Use in LabVIEW Advanced Settings Window
The remote computer is running the LabVIEW development system or a LabVIEW executable, and you are calling a VI that is not a part of a LabVIEW project.	Use the Remove VI Path control to specify the path to the VI on the remote computer.

Scenario	Control to Use in LabVIEW Advanced Settings Window
You are calling a VI that is part of a LabVIEW project.	Use the Remote Project Path control to specify the path to the LabVIEW project on the remote computer, and use the Remote VI Path control to specify the path within the LabVIEW project to the VI you want to call.

Downloading VIs to a PXI Controller

You can use the LabVIEW development system to download VIs to a PXI controller that runs the LabVIEW Real-Time Module. You can also use the TestStand FTP Files step type to download files from and upload files to a remote computer.

Complete the following steps to use FTP to download VIs to the PXI controller.

1. Use the LabVIEW development system to create a Source Distribution with all the VIs to ensure that you include all the dependencies of the VIs to transfer to the hard drive of the LabVIEW Real-Time (RT) target.
2. In the distribution, disable the options to exclude VIs from the `<LabVIEW>\vi.lib`, `<LabVIEW>\instr.lib`, and `<LabVIEW>\user.lib` directories.
3. Use FTP to transfer the source distribution output to the hard drive of the LabVIEW RT target.

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



Note

- TestStand does not support remotely calling VIs that use projects on PXI controllers that run the LabVIEW Real-Time Module because LabVIEW RT does not support LabVIEW projects.
- Refer to the Getting Started with the LabVIEW Real-Time Module manual (`RT_Getting_Started.pdf`) in the

<LabVIEW>\manuals directory for more information about downloading VIs to a PXI controller.

Related concepts:

- [Calling VIs on Remote Computers](#)
- [Organizing Test Program Files with LabVIEW Projects](#)

Configuring the LabVIEW VI Server to Run VIs Remotely

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

You can configure the VI Server settings for the following test system configurations:

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

Configuring Additional VI Server Options

Use the following sections in the Options dialog box in LabVIEW to further configure a VI Server:

- **VI Server category**—Enable the TCP/IP protocol and the VI call options. You can also specify the TCP/IP port the server uses. The port you specify in LabVIEW must

be the same port you specify in TestStand in the LabVIEW Advanced Settings window.

- **Machine Access section of the VI Server category**—Allow specific computers access to the VI Server. You can also specify certain computers or entire domains that can call VIs on the server computer.
- **Exported VIs section of the VI Server category**—Configure the VIs you want to call through the VI Server. You must export all VIs you want to call remotely from TestStand. The default setting in LabVIEW is to export all VIs. The asterisk (*) in the Exported VIs list control indicates this setting.

Configuring VI Server Security

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

Related concepts:

- [Calling VIs on Remote Computers](#)

TestStand Example Programs

TestStand includes a variety of example programs that you can use to help you learn key concepts or to serve as a starting point for applications you create.

You can browse the examples in the following ways:

- Open the `Examples.tsw` workspace file located in the `<TestStand Public>\Examples` directory.
- Open an example sequence file from the subdirectories of the `<TestStand Public>\Examples` directory.

Related concepts:

- [TestStand Directory Structure](#)

Related information:

- [TestStand Featured Examples](#)

Examples for Built-In Step Types

The Built-In Step Types examples demonstrate how to use the default step types provided in the TestStand insertion palette. This directory includes the following example programs.

Database Step Types

Purpose

This example demonstrates how to use the Database step types to delete and create tables in a database, as well as how to modify the records in the database.

Example File Location

`<TestStand Public>\Examples\Built-In Step Types\Database`

Step Types\Database Step Types.seq

Highlighted Features

Database step types

Major API

None

Prerequisites

(32-bit TestStand) You must have the Microsoft Jet 4.0 Object Linking and Embedding Database (OLE DB) Provider installed. (64-bit TestStand) You must have the Microsoft Access Database Engine 2010 Redistributable installed. Visit ni.com/info and enter the Info Code 64TSacddb to access the NI support article, Using Microsoft Access Databases with 64-bit TestStand, for more information about installing this provider.

How to Use This Example

Complete the following steps to use this example.

1. Review the following steps in the `MainSequence`:
 - a. The Find Access MDB File step, which is a Statement step, attempts to locate the Microsoft Access database file (`.mdb`) and store its path in the `MDBFilePath` local variable.
 - b. The Open Database step opens the database for use in TestStand, returns a handle, and stores the handle in the `DatabaseHandle` local variable.
 - c. The Drop Table step, which is an Open SQL Statement step, drops the `TEST_TABLE` database table, if it exists.
 - d. The Create Table step, which is an Open SQL Statement step, creates the `TEST_TABLE` table.
 - e. The Open SQL Statement step selects all columns in the `TEST_TABLE` table.
 - f. The SQL Action - Set Values step, which is a Data Operation step, adds data to the `TEST_TABLE` table and loops 10 times because the Goto step that follows the SQL Action - Set Values step targets the Start Loop step, which is a Label step, and specifies a post action and precondition that starts and stops the loop.

- g. The Start Loop 2 step starts a second loop that uses Data Operation steps to obtain the values and strings written to the `TEST_TABLE` table.
 - h. The Cleanup step group contains Close SQL Statement and Close Database steps.
2. Select **Execute»Single Pass** to run the sequence.
 3. When execution completes, select **Tools»Database Viewer** to launch the TestStand Database Viewer application.
 4. Select **File»Open**, browse to `<TestStand Public>\Examples\Database\Access.mdb`, and click **OK**.
 5. Right-click the `TEST_TABLE` table and select **View Data** from the context menu to display the data contained in the `TEST_TABLE` table.

Related concepts:

- [Databases and Tables](#)
- [TestStand Directory Structure](#)
- [Step Groups](#)

Flow Control Step Types

Purpose

This example demonstrates using the Flow Control step types to control sequence flow. These step types include If, Else If, Else, For, For Each, While, Do While, Select, Case, Break, Continue, and End.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Flow
Control Step Types\Flow Control Step Types.seq
```

Highlighted Features

Flow Control step types

Major API

None

Prerequisites

None

How to Use This Example

The example sequence file demonstrates several methods for controlling sequence flow.

If, Else If, and End

The Prompt for option for If/Else/Else step prompts you to click one of three buttons, and each launches its own message popup that indicates the button you clicked. The If step determines whether you clicked the **Option A** button. If the step evaluates to `True`, the sequence executes the steps that immediately follow the If step. If the step evaluates to `False`, the sequence continues to the first Else step, which in this case is an Else If step.

The Else If step behaves the same as the If step and determines whether you clicked the **Option B** button. If the step evaluates to `True`, the sequence continues executing until it encounters another Else step or encounters an End step. If the Else If step evaluates to `False`, the sequence executes the next Else step or continues to an End step if no additional Else steps exist.

When a sequence encounters an Else step, the sequence always executes all steps that follow the Else step. You can only use one Else step for each If step, but you can use multiple Else If steps.

Select and Case

The Prompt for option for Select/Case step prompts you to click one of three buttons, and each launches its own message popup that indicates the button you clicked. The Select step determines which button you clicked. Two or more Case steps always follow a Select step. You can specify multiple values for a single case using an array, such as `{"y","yes"}`. All the steps contained in a Case block execute, depending on the value the Select step evaluates. The End steps specify the end of a Case block or a Select structure.

For

The For step executes a block of steps multiple times. The For Loop edit tab on the Step Settings pane specifies that the For step begins with the `Locals.index` variable set to 0. The loop continues until the value of the `Locals.index` variable is greater than or equal to the value you specify. Each time the For step executes, the `Locals.index` variable increments by 1. Therefore, the Beep at done determined by loop index step in the For block executes the specified number of times and uses the increasing index.

For Each

The For Each step behaves like the For step but iterates over every element in an array rather than iterate depending on a specified condition. The `Locals.CurrentMembers` variable is an array that specifies three elements, so the steps in the For Each block execute three times. The current element that the loop is iterating on is stored in the `Locals.CurrentName` property. This setting is specified in the ForEach loop tab of the step settings of the For Each step.

Do While

The Do While step defines a block of steps that always execute at least once. The DoWhile Loop edit tab on the Step Settings pane specifies an expression that the Do While step evaluates after all the steps in the Do While block execute. If the expression evaluates to `True`, the steps in the Do While block execute again. If the expression evaluates to `False`, the sequence continues to the next step.

While

The While step defines a block of steps that execute when an expression specified on the While Loop edit tab on the Step Settings pane evaluates to `True`. If the expression evaluates to `False`, the sequence continues to the next step. The While loop differs from the Do While loop because the While loop evaluates its expression before each iteration and not after each iteration. Therefore, the steps in the While block might never execute.

Break and Continue

The Break step terminates a For, For Each, While, or Do While loop or a Case block. When a sequence encounters a Break step, the sequence exits the block and continues to the next step.

The Continue step terminates the current iteration of a For, For Each, While, or Do While loop and prompts the sequence to begin the next iteration of the loop.

In addition to Flow Control step types, you can also use preconditions or post actions step properties to control sequence flow.

An advantage of using step properties is that the sequence can use fewer steps. In addition, the flow control logic is attached to the step. For example, if you specify a precondition of `Locals.PowerSupplyOn == True` for a step, the precondition remains attached to the step even if you move or copy the step to a new location. A disadvantage is that the sequence flow logic is not immediately visible when viewing the sequence. You can make flow control more visible by using preconditions on separate Goto steps and using Label steps as branch targets. However, the functionality you create in this way, such as loops or conditionally-executed blocks of steps, is embedded in the precondition expressions and branch targets, and requires anyone who reads the sequence to review the preconditions and branch targets to understand the flow control.

An advantage of using separate Flow Control steps is that the flow control logic is visible in the sequence it is expressed in separate steps that identify the flow control operations the steps perform. In addition, applications typically display the flow control structures you create with Flow Control steps with indentation and grouping bars that improve the readability of the sequence. Furthermore, because Flow Control steps operate on blocks of steps demarcated with separate End steps, you do not need to specify specific steps as branch destinations, so you can more easily edit and rearrange steps because you do not need to update branch destinations. You can use step properties, separate Flow Control steps, or both, depending on which functionality best suites the requirements of an application.

Related concepts:

- [TestStand Directory Structure](#)

IVI Step Types

Purpose

This example demonstrates how to use IVI step types to configure and take measurements from IVI instruments with the following logical names: SampleDMM, SampleScope, SampleFGen, SamplePowerSupply, and SampleSwitch. This example contains Setup steps that initialize the logical names in simulation mode and prevent interactive simulation for measurement values. Visit ni.com/ivi for more information about IVI.



Note As of TestStand 2019, the TestStand IVIStepTypes package is not included in the default TestStand ISO image. It is now included in the `AdditionalInstallers` folder. You can still install this package using the TestStand offline installer or NI Package Manager.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\IVI Step Types\IVI Step Types.seq
```

Highlighted Features

- IVI step types
- NI Session Manager

Major API

None

Prerequisites

Complete the following steps to install the required software and IVI-compliant instrument drivers.

1. Install the following additional components from the NI Device Driver DVD that comes with TestStand:
 - a. NI Measurement & Automation Explorer (MAX)

- b. IVI Compliance Package (ICP)
- 2. Refer to the Instrument Driver Network at ni.com/idnet to download and install IVI-compliant instrument drivers for each of the IVI classes this example uses. Search for the manufacturer or type of instrument you want to download. Some basic IVI-compliant instrument drivers you can use with this example include the following:
 - **Dmm**—hp34401a
 - **Scope**—tkds30xx
 - **Fgen**—hp33120a
 - **Switch**—age1442a
 - **PowerSupply**—hp66xxbc

In MAX, the driver installer automatically creates a driver session entry under the **Driver Sessions** item and populates the software module information under the **Instrument Driver Software Module** item in the *Advanced* folder.

- 3. Complete the following steps to create a logical name for each instrument driver.
 - a. In MAX, expand the **IVI Drivers** category to the left.
 - b. Right-click the **Logical Names** item in the configuration tree and select **Create New** from the context menu.
 - c. Complete the configuration panel for the new logical name. Logical names are case-sensitive.



Note Class drivers display simulation dialog boxes if interactive simulation is enabled. These dialog boxes might not have taskbar items and can go behind the TestStand Sequence Editor. In addition, the IVI soft front panels remain visible until you close the Execution window.

How to Use This Example

This example communicates with each type of instrument driver, one after the other. In each case, the sequence configures the instrument and takes a measurement. The sequence contains Label steps denote the beginning of steps that communicate with each type of IVI instrument.

Review the edit tabs on the Step Settings pane for each IVI step to see how the step configures the instrument and takes measurements.

TestStand stores the logical names the sequence expects as file global variables. Each

variable name must match the logical name created for each driver in MAX.

Related concepts:

- [TestStand Directory Structure](#)
- [Session Manager](#)

Related tasks:

- [Installing TestStand](#)

Message Popup Step Type - Automatically Dismissing

Purpose

This example demonstrates the following two approaches for creating Message Popup steps that automatically dismiss after a condition is met:

- Dismissing the message after a set amount of time
- Dismissing the message after a set condition is met, such as detecting that a new UUT has been attached to the test fixture

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Message  
Popup Step Type\Message Popup Step Type - Automatically  
Dismissing.seq
```

Highlighted Features

- Message Popup Timeout
- Creating New Executions

Major API

Execution.Terminate

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequences and steps in this example:

1. Select the **UUT Detected** step. In the Step Settings pane, select the **Text and Buttons** tab.
2. Observe that Button 1 is specified in the Timeout Button field, and the Time to wait field is set to three seconds. This setting configures the step to automatically select Button 1 and dismiss the message box after three seconds.
3. Select the **Prompt for User Action** sequence. Observe that this sequence creates a new execution to launch the dialog. The sequence will dismiss the dialog by terminating the execution.



Note The Launch Dialog in new Execution step stores a reference to the new execution in the `Locals.messageExecution` variable using the Sequence Call Advanced Settings dialog box.

4. After launching the dialog, the sequence simulates polling for a UUT. In a real application, this section would typically call a code module which returns once the condition is met
5. Once the condition is met, the `Execution.Terminate` method of the TestStand API terminates the new execution, thereby dismissing the message popup.

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to run the sequence.
2. Observe that the first message popup dismisses after three seconds.
3. Observe that the second message popup step dismisses once the Check for UUT condition is met.

Related concepts:

- [TestStand Directory Structure](#)

- [Executions](#)

Message Popup Step Type - Get User Input

Purpose

This example demonstrates how to use Message Popup steps to obtain user input using the following two methods:

- Using Buttons
- Using a Text Field

The example determines what actions to take based on user selections.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Message  
Popup Step Type\Message Popup Step Type - Get User  
Input.seq
```

Highlighted Features

- Case steps
- Error handling
- Message Popup steps
- Preconditions
- Select steps

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to review the steps in the Buttons sequence. This

sequence demonstrates how you can use a message popup step to prompt a user for input using buttons.

1. Select the **Board Test Selection – Buttons** step. In the Step Settings pane, select the **Text and Buttons** tab.
2. Observe that the button names are specified in the Button 1 through Button 5 fields. The message popup step will display all buttons with specified names.
3. The example uses a Select statement to respond to the user selection. The Select statement accesses the `RunState.Sequence.Main["Board Test Selection – Buttons"].Result.ButtonHit` property to check the button the user chooses.

Complete the following steps to review the steps in the PromptForInput sequence. This sequence demonstrates how you can use a message popup step to prompt a user for input using a response text box.

1. Select the **Select Board to Test - Text Field** step. In the Step Settings pane, select the **Options** tab.
2. Observe that the **Enable Response Text Box** option is enabled. This option adds a response text box to the message popup to receive user input.
3. The example uses a Select statement to respond to the user selection. The Select statement uses the expression `Val(RunState.Sequence.Main["Select Board To Test - Text Field"].Result.Response)` to check the response the user enters.
4. If an invalid value is entered, the default case notifies the user of the response. This step triggers a post action, in which the Select Board To Test - Text Field step executes again to allow the user to enter a valid response.



Note The `Val()` expression function converts the text response string to a number.

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to run the sequence.
2. When prompted, click a button or enter a value. Notice that the sequence responds differently based on the input you provide.

Related concepts:

- [TestStand Directory Structure](#)

Multiple Numeric Limit Test Step Type

The <TestStand Public>\Examples\Built-In Step Types\Multiple Numeric Limit Test Step Type directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Multiple Numeric Limit Test Step Type - LabWindows/CVI

Purpose

This example demonstrates how to use a Multiple Numeric Limit Test step with a LabWindows/CVI code module.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Multiple  
Numeric Limit Test Step Type\CVI\Multiple Numeric Limit  
Step Type.seq
```

Highlighted Features

Multiple Numeric Limit Test steps

Major API

None

Prerequisites

You must have the LabWindows/CVI Run-Time Engine installed and you must configure the LabWindows/CVI Adapter to execute steps in-process. If you want to use the

Execute Steps in an External Instance of CVI option, you must have the LabWindows/CVI development environment installed.

How to Use This Example

This example contains four Multiple Numeric Limit Test steps. The first three steps obtain four measurement values and compare them to the following high and low values specified on the Limits tab of the Step Settings pane:

Measurement	Comparison	Low Value	High Value
VOLTAGE 0	GELE ($\geq \leq$)	3	5
VOLTAGE 1	GELE ($\geq \leq$)	3	5
FREQUENCY	GELE ($\geq \leq$)	110	125
RMS	GELE ($\geq \leq$)	0.5	1.0

Each step demonstrates a different way of using the Multiple Numeric Limit Test step functionality.

- The first step, Multiple Numeric Limit Test 1, calls a code module that inserts the measurement values one element at a time into the `Step.NumericArray` property. For example, the step inserts the value for the `VOLTAGE 0` measurement in `Step.NumericArray[0]`, inserts the value for the `VOLTAGE 1` measurement in `Step.NumericArray[1]`, and so on, as shown on the Data Source tab of the Step Settings pane.
- The second step, Multiple Numeric Limit Test 2, calls a code module that writes a single numeric array to the `Step.NumericArray` property instead of writing the four array elements individually.
- The third step, Multiple Numeric Limit Test 3, uses the <None> Adapter and therefore does not call a code module. The Data Source tab of the Step Settings pane for this step shows that the step obtains measurement values from local variables such as `Locals.Voltage0`, `Locals.Voltage1`, and so on instead of from the `Step.NumericArray` property. The Statement steps in the Setup step group use expressions to assign these variables.
- The fourth step, Multiple Numeric Limit Test 4, is similar to the third step, but demonstrates the new `EQ+/-` comparison type supported by TestStand 2016 and

later. See the table below for more information about this step's threshold values.

Measurement	Comparison	Threshold	Nominal Value	Lower Threshold	Upper Threshold	Computed Low Limit	Computed High Limit
VOLTAGE 0	EQ (==+/-)	Percentage	4	25	25	3	5
VOLTAGE 1	EQ (==+/-)	PPM	4	250000	250000	3	5
FREQUENCY	EQ (==+/-)	Delta Value	115	5	10	110	125
RMS	EQ (==+/-)	Delta Value	0.75	0.25	0.25	0.5	1

For a Multiple Numeric Limit Test step to pass, every test in the step must pass. You can try changing a limit to cause a test to fail, but ensure that you do not save any changes to the sequence file before executing the sequence or closing the sequence file.

Related concepts:

- [TestStand Directory Structure](#)
- [Step Groups](#)

Multiple Numeric Limit Test Step Type - LabVIEW

Purpose

This example demonstrates how to use a Multiple Numeric Limit Test step with a LabVIEW code module.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Multiple  
Numeric Limit Test Step Type\LabVIEW\Multiple Numeric Limit  
Step Type.seq
```

Highlighted Features

Multiple Numeric Limit Test steps

Major API

None

Prerequisites

You must have the LabVIEW development system installed and you must configure the LabVIEW Adapter to use the LabVIEW development system.

How to Use This Example

This example contains four Multiple Numeric Limit Test steps. The first three steps obtain four measurement values and compare them to the following high and low values specified on the Limits tab of the Step Settings pane:

Measurement	Comparison	Low Value	High Value
VOLTAGE 0	GELE ($\geq \leq$)	3	5
VOLTAGE 1	GELE ($\geq \leq$)	3	5
FREQUENCY	GELE ($\geq \leq$)	110	125
RMS	GELE ($\geq \leq$)	0.5	1.0

Each step demonstrates a different way of using the Multiple Numeric Limit Test step functionality.

- The first step, Multiple Numeric Limit Test 1, calls a code module that inserts the measurement values one element at a time into the `Step.NumericArray` property. For example, the step inserts the value for the `VOLTAGE 0` measurement in `Step.NumericArray[0]`, inserts the value for the `VOLTAGE 1` measurement in `Step.NumericArray[1]`, and so on, as shown on the Data Source tab of the Step Settings pane.
- The second step, Multiple Numeric Limit Test 2, calls a code module that writes a single numeric array to the `Step.NumericArray` property instead of writing the four array elements individually.
- The third step, Multiple Numeric Limit Test 3, uses the `<None>` Adapter and therefore does not call a code module. The Data Source tab of the Step Settings pane for this step shows that the step obtains measurement values from local

variables such as `Locals.Voltage0`, `Locals.Voltage1`, and so on instead of from the `Step.NumericArray` property. The Statement steps in the Setup step group use expressions to assign these variables.

- The fourth step, Multiple Numeric Limit Test 4, is similar to the third step, but demonstrates the new `EQ+/-` comparison type supported by TestStand 2016 and later. See the table below for more information about this step's threshold values.

Measurement	Comparison	Threshold	Nominal Value	Lower Threshold	Upper Threshold	Computed Low Limit	Computed High Limit
VOLTAGE 0	EQ (==+/-)	Percentage	4	25	25	3	5
VOLTAGE 1	EQ (==+/-)	PPM	4	250000	250000	3	5
FREQUENCY	EQ (==+/-)	Delta Value	115	5	10	110	125
RMS	EQ (==+/-)	Delta Value	0.75	0.25	0.25	0.5	1

For a Multiple Numeric Limit Test step to pass, every test in the step must pass. You can try changing a limit to cause a test to fail, but ensure that you do not save any changes to the sequence file before executing the sequence or closing the sequence file.

Related concepts:

- [TestStand Directory Structure](#)
- [Step Groups](#)

Multiple Numeric Limit Test Step Type - .NET

Purpose

This example demonstrates how to use a Multiple Numeric Limit Test step with a .NET assembly code module.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Multiple
Numeric Limit Test Step Type\DotNET\Multiple Numeric Limit
```

Step Type.seq

Highlighted Features

Multiple Numeric Limit Test steps

Major API

None

Prerequisites

None

How to Use This Example

This example contains four Multiple Numeric Limit Test steps. The first three steps obtain four measurement values and compare them to the following high and low values specified on the Limits tab of the Step Settings pane:

Measurement	Comparison	Low Value	High Value
VOLTAGE 0	GELE ($\geq \leq$)	3	5
VOLTAGE 1	GELE ($\geq \leq$)	3	5
FREQUENCY	GELE ($\geq \leq$)	110	125
RMS	GELE ($\geq \leq$)	0.5	1.0

Each step demonstrates a different way of using the Multiple Numeric Limit Test step functionality.

- The first step, Multiple Numeric Limit Test 1, calls a code module that inserts the measurement values one element at a time into the `Step.NumericArray` property. For example, the step inserts the value for the `VOLTAGE 0` measurement in `Step.NumericArray[0]`, inserts the value for the `VOLTAGE 1` measurement in `Step.NumericArray[1]`, and so on, as shown on the Data Source tab of the Step Settings pane.
- The second step, Multiple Numeric Limit Test 2, calls a code module that writes a

single numeric array to the `Step.NumericArray` property instead of writing the four array elements individually.

- The third step, Multiple Numeric Limit Test 3, uses the <None> Adapter and therefore does not call a code module. The Data Source tab of the Step Settings pane for this step shows that the step obtains measurement values from local variables such as `Locals.Voltage0`, `Locals.Voltage1`, and so on instead of from the `Step.NumericArray` property. The Statement steps in the Setup step group use expressions to assign these variables.
- The fourth step, Multiple Numeric Limit Test 4, is similar to the third step, but demonstrates the new `EQ+/-` comparison type supported by TestStand 2016 and later. See the table below for more information about this step's threshold values.

Measurement	Comparison	Threshold	Nominal Value	Lower Threshold	Upper Threshold	Computed Low Limit	Computed High Limit
VOLTAGE 0	EQ (==+/-)	Percentage	4	25	25	3	5
VOLTAGE 1	EQ (==+/-)	PPM	4	250000	250000	3	5
FREQUENCY	EQ (==+/-)	Delta Value	115	5	10	110	125
RMS	EQ (==+/-)	Delta Value	0.75	0.25	0.25	0.5	1

For a Multiple Numeric Limit Test step to pass, every test in the step must pass. You can try changing a limit to cause a test to fail, but ensure that you do not save any changes to the sequence file before executing the sequence or closing the sequence file.

Related concepts:

- [TestStand Directory Structure](#)
- [Step Groups](#)

Property Loader Step Type - Loading Limits from a Database

Purpose

This example demonstrates how to use the Property Loader step type to load step properties, such as limits for Numeric Limit Test steps, from a database.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Property  
Loader Step Type\Property Loader - Loading Limits From a  
Database.seq
```

Highlighted Features

- Database step types
- Database Viewer Application
- Property Loader steps

Major API

None

Prerequisites

TestStand loads the property values from the `Access.mdb` file located in the same directory as the sequence file. (32-bit TestStand) You must have the Microsoft Jet 4.0 Object Linking and Embedding Database (OLE DB) Provider installed. (64-bit TestStand) You must have the Microsoft Access Database Engine 2010 Redistributable installed. Visit ni.com/info and enter the Info Code 64TSacddb to access the NI support article, Using Microsoft Access Databases with 64-bit TestStand, for more information about installing this provider.

How to Use This Example

This example uses a Property Loader step to load property values, including local variable values and step limits, from a Microsoft Access database file. The Property Loader step applies these values to the sequence before the test steps execute, which demonstrates how the limits and variables can be loaded each time a sequence executes.

Complete the following steps to review the Property Loader step settings.

1. On the Steps pane, select the **Property Loader** step in the Setup step group.
2. On the Step Settings tab of the Steps pane, review the items in the sources table. In

this table, the database is configured as the properties source.

3. On the Target File and Source Settings tab, review the properties selected in the Property Tree control. The step loads the `Comp`, `Limits.Low`, and `Limits.High` values for each step specified, as well as a local and file global variable.

Complete the following steps to run this example.

1. Select **Execute»Single Pass** to run the sequence.
2. When execution completes, review the report on the Report pane. The report indicates that the Video step failed because the test result of 65 was not within the acceptable limits.
3. Complete the following steps to edit the database to include an acceptable value.
 - a. Select **Tools»Database Viewer** to launch the TestStand Database Viewer application.
 - b. Select **File»Open**, browse to `<TestStand Public>\Examples\Built-In Step Types\Property Loader Step Type\Access.mdb`, and click **OK**.
 - c. Right-click the `STEP_PROPERTIES` table in the Database Explorer pane and select **View Data**.
 - d. Right-click the table and select **Edit Data**.
 - e. Change the value of the `LIMITS_LOW` field for the video record from 71 to 60. Click the **Submit** button to submit the values to the database.
 - f. Close the Database Viewer application.
4. Select **Execute»Single Pass** to run the sequence again.
5. When execution completes, review the report on the Report pane. The report indicates that the Video step passed because the test result is now within the acceptable limits.

Related concepts:

- [TestStand Directory Structure](#)
- [Step Groups](#)

Property Loader Step Type - Loading Limits from a File Containing Alias

Purpose

This example demonstrates using the Property Loader step type to load the properties from a source containing alias names instead of property lookup strings. These aliases and corresponding properties are defined in a separate .pla file. The example loads properties at the beginning of the MainSequence.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Property  
Loader Step Type\ Property Loader - Loading Limits From  
File Containing Alias.seq
```

Highlighted Features

- Property Loader steps
- ReportOptions callback

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequence functionality.

1. On the Steps pane, select the **Property Loader** step in the Setup step group.
2. On the Target File and Source Settings tab, observe that the file `AliasNames.pla` is specified for the Alias Location field. This file specifies alias names for various properties, making it easier to identify them in a limits file. Open the file in a text editor to review the contents.

3. On the Step Settings tab in the Step Settings pane, click the **View Source File** button in the sources table to open the properties file. Observe that the properties in the file are specified using the aliases defined in the alias file.

Complete the following steps to run this example.

1. Select **Execute»Single Pass** to run the sequence.
2. When execution completes, review the report on the Report pane. Ensure that the limits were changed based on the property loader step configuration.

Related concepts:

- [TestStand Directory Structure](#)

Property Loader Step Type - Loading Limits from a Microsoft Excel File

Purpose

This example demonstrates how to use a Property Loader step type to load limits from a Microsoft Excel spreadsheet file. This example uses the LabWindows/CVI Adapter for Numeric Limit Test steps.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Property  
Loader Step Type\Property Loader - Loading Limits From  
Excel.seq
```

Highlighted Features

- Property Loader steps
- ReportOptions callback

Major API

None

Prerequisites

You must have Microsoft Excel installed.

How to Use This Example

The Property Loader step starts reading data from the START LIMITS marker and reads data until it encounters the END LIMITS marker.

Complete the following steps to review the sequence functionality.

1. On the Steps pane, select the **Property Loader** step in the Setup step group.
2. On Target File and Source Settings tab in the step settings pane, review the properties selected in the Property Tree control. The step loads the `Limits.Low` and `Limits.High` values for each step.
3. On the Step Settings tab of the Steps pane, click the **View File** button in the sources table to open `<TestStand Public>\Examples\Built-In Step Types\Property Loader Step Type\Limits.xls` in Excel. The columns in the Excel file specify the properties to import, and the rows specify the names of the steps in the sequence.

Complete the following steps to run this example.

1. Select **Execute»Single Pass** to run the sequence.
2. When execution completes, review the report on the Report pane. The report indicates that TestStand updated the limits of the steps based on the values in the Excel file.
3. In the `Limits.xls` file, change the Flow Rate value from 1 to 1.1 and save the file.
4. Select **Execute»Single Pass** to run the sequence again.
5. When execution completes, review the report on the Report pane. The report indicates that the results have changed because of the updated step limits.

Related concepts:

- [Programming with the TestStand API in LabWindows/CVI](#)
- [TestStand Directory Structure](#)
- [Step Groups](#)

Property Loader Step Type - Loading Limits for Multiple Sequences

Purpose

This example demonstrates using the Property Loader step type to load the properties for multiple sequences in a client sequence file. The example loads properties at the beginning of the MainSequence.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Property  
Loader Step Type\Property Loader - Loading Limits For  
Multiple Sequences.seq
```

Highlighted Features

Property Loader steps

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to change properties that already exist in a sequence.

1. Select the **MainSequence**.
2. On the Steps pane, select the **Property Loader** step.
3. On the Step Settings pane, click the Step Settings tab. Click the **View File** button in the sources table to open `propertyData.txt`, which is located in the same directory as the example sequence file and is the file the Property Loader step is using to specify property values. The text file specifies a data block for each sequence in the sequence file so the step can load different values for the same

property in different sequences. The Start and End of Data markers for the data block include the name of the sequence. In the text file, the `Properties for Sequence 1` section specifies the limits for the Numeric Limit Test step as 15 for `Limits.Low` and 20 for `Limits.High`. The text file contains very specific formatting to which you must adhere for the Property Loader step to function correctly. The best method for generating a file for use with a Property Loader step is to specify the properties in the sequence file and then select **Tools»Import/Export Properties** to launch the Import/Export Properties dialog box, in which you can generate a file that a Property Loader step can use.

4. Select **Execute»Single Pass** to run the sequence.
5. When execution completes, review the report on the Report pane. Notice that the limit values in each subsequence have changed to the values specified by the property file.

Related concepts:

- [TestStand Directory Structure](#)

Property Loader Step Type - Loading Limits from Multiple Sources

Purpose

This example demonstrates using the Property Loader step type to load data from multiple property loader sources. The example loads properties from `GenericLimits` to load limits for Voltage and Temperature and loads properties from `SpeacializedLimits` to override the limits for Voltage.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Property
Loader Step Type\ Property Loader - Loading Limits From
Multiple Sources.seq
```

Highlighted Features

- Property Loader steps

- ReportOptions callback

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequence functionality.

1. On the Steps pane, select the **Property Loader** step in the Setup step group.
2. On the Step Settings tab in the Step Settings pane, observe that two source locations are specified in the source table. This indicates that properties will be loaded from multiple sources, in this case two text files.
3. Navigate to the <TestStand Public>\Examples\Built-In Step Types\Property Loader Step Type directory and review the GenericLimits.txt and SpecializedLimits.txt files. Observe that both files have properties defined for the voltage test. In this case, the file specified last in the source table will take precedence.

Complete the following steps to run this example.

1. Select **Execute»Single Pass** to run the sequence.
2. When execution completes, review the report on the Report pane. Ensure that the limits were changed based on the property loader step configuration.

Related concepts:

- [TestStand Directory Structure](#)

Property Loader Step Type - Loading Limits from Two Different Text Files

Purpose

This example demonstrates how to use the Property Loader step to dynamically load step limits from different text files depending on user input.

In the Setup step group of the `MainSequence`, the Message Popup step launches a dialog box to prompt the user to choose which file TestStand uses to load limits values. The Select Limit File Statement step in the Setup step group selects the limit file. The expression for this step uses a conditional operation in which TestStand loads limits from `RevisionALimits.txt` when the value of the `Result.ButtonHit` property equals 1. The expression specifies that TestStand loads limits from `RevisionBLimits.txt` for all other cases.

When you write a sequence in which the user must choose between three or more limit files, use a Property Loader step for each limit file and use a precondition for each step that checks for user input. For example, when the Message Popup step specifies four buttons, each Property Loader step can check which button was clicked in the precondition.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Property  
Loader Step Type\Property Loader - Loading Limits From  
Different Files.seq
```

Highlighted Features

- Property Loader steps
- ReportOptions callback

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequence functionality.

1. On the Steps pane, select the **Property Loader** step in the Setup step group.
2. On the Step Settings tab of the Step Settings pane, confirm that the source location in the sources table specifies the `Locals.LimitFileToLoad` variable. The value of this variable at run time determines which file TestStand loads.
3. Navigate to the `<TestStand Public>\Examples\Built-In Step Types\Property Loader Step Type` directory and review the `RevisionALimits.txt` and `RevisionBLimits.txt` files. Each file specifies a different set of limit values for the sequence.
4. On the Steps pane, select the **Select Limit File** step, which is an instance of a Statement step type. This step sets the `Locals.LimitFileToLoad` variable to the correct value depending on which file the user selects.

Complete the following steps to run this example.

1. Select **Execute»Single Pass** to run the sequence.
2. When prompted, select the **Revision A Limits** option.
3. When execution completes, review the report on the Report pane. The report indicates that all steps passed.
4. Select **Execute»Single Pass** to run the sequence again.
5. When prompted, select the **Revision B Limits** option.
6. When execution completes, review the report on the Report pane. The report indicates that not all steps passed because of the different set of limit values. The step results have not changed.

Related concepts:

- [Step Groups](#)
- [TestStand Directory Structure](#)

Statement Step Type

Purpose

This example demonstrates how to use an expression in a Statement step to dynamically change the high and low limits of a Numeric Limit Test step.



Note NI typically recommends that you use expressions when you configure a Numeric Limit Test step or to use a Property Loader step to change limits, but this example demonstrates how to change step properties.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\Statement  
Step Type\Statement Step Type.seq
```

Highlighted Features

Expressions

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane, select the **Numeric Limit Test** step.
2. On the Step Settings pane, click the Limits tab. The comparison type for the step is currently GELE ($\geq$ $\leq$), with a high value of 10 and a low value of 0. When accessed from the Numeric Limit Test step, the numeric limits for the Numeric Limit Test step are `Step.Limit.High` and `Step.Limit.Low`. However, another step cannot access the variables `Step.Limit.High` and

`Step.Limit.Low`, as shown in this example. Instead, you must explicitly specify the step that specifies the properties you want to access.

3. On the Steps pane, select the **A Statement that changes the numeric limits** step, which is a Statement step.
4. On the Step Settings pane, click the Expression edit tab. The Expression tab contains two expressions, separated by a comma, that specify the high limit as 11 and the low limit as 0: `RunState.Sequence.Main["Numeric Limit Test"].Limits.High = 11, RunState.Sequence.Main["Numeric Limit Test"].Limits.Low = 0` The expressions specify that you want to use the Numeric Limit Test step to specify the properties you want to access. Alternatively, you could use the following expression:
'`RunState.NextStep.Limits.High`'
5. Select **Execute»Single Pass** to run the sequence.
6. When the execution completes, review the report. For the Numeric Limit Test step, the High Limit value is 11, and the Low Limit value is 0.

Because the Statement step executed before the Numeric Limit Test step in the sequence, changing the limits after the Numeric Limit Test executes does not change the limit values.

Related concepts:

- [TestStand Directory Structure](#)
- [Expressions](#)

Synchronization Step Types - Auto Schedule

Purpose

This example demonstrates the Auto Schedule and Use Auto Scheduled Resource step types. These step types are used to dynamically determine the order in which a UUT uses its required resources to minimize delays that occur while the UUT waits for a resource to become available.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\
Synchronization Step Types\Synchronization Step Types -
```

Auto Schedule.seq

Highlighted Features

- Auto Schedule step type
- Use Auto Scheduled Resource step type
- Batch process model

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequences and steps in the example.

1. On the Sequences pane, select the **MainSequence**. The Auto Schedule step at the beginning of the test defines a block of Use Auto Scheduled Resource sub-blocks. When the test sequence executes, each test socket executes each Use Auto Scheduled Resource sub-block once. The order in which the sub-blocks execute can vary between test sockets to improve execution time.
2. Select one of the Use Auto Scheduled Resource steps and navigate to the Auto Scheduled Resource Settings Edit Tab on the Step Settings pane. Notice that at least one resource name is defined in the Resource Lock Alternatives list. The UUT does not execute the test steps in this Use Auto Scheduled Resource sub-block until it has exclusive access to the resources in this list.
3. Select each of the Use Auto Scheduled Resource steps to compare the settings for the Resource Lock Alternatives. You can specify multiple resource lock alternatives to indicate that the sub-block can execute if any of the alternatives are available. To lock access to all of these resources before executing the sub-block, use an array of lock names to specify that a section must lock multiple resources. In this case, no locks are reserved unless all resources are available.

Complete the following steps to run the example.

1. Select **Execute»Single Pass** to run the sequence.
2. A prompt appears that describes the Resource Usage Profiler, which is a tool to monitor the status of synchronization objects in a test sequence. You can use this tool to observe the execution order of test steps in a test sequence with Auto Schedule steps. You can choose to launch the tool to view additional details during the test.
3. As the test sequence executes, dialog boxes indicate the current status of the tests. Notice that each UUT is only able to execute a Use Auto Scheduled Resource sub-block if it is able to gain exclusive access to the required resources.
4. Once execution completes, you can examine the test report for each UUT to notice the order of execution. The Auto Schedule steps do not guarantee a specific execution order for each test step. The order can vary among test sockets and among executions. You can run the test sequence additional times to compare the execution order among different test runs.

Related concepts:

- [TestStand Directory Structure](#)
- [Batch Process Model](#)

Synchronization Step Types - Batch Synchronization

Purpose

This example demonstrates the Batch Synchronization step types and step settings available when you use the Batch process model. Use Batch synchronization to coordinate test operations among all test sockets in a batch and to execute test steps selectively, concurrently, or sequentially.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\  
Synchronization Step Types\Synchronization Step Types -  
Batch Synchronization.seq
```

Highlighted Features

- Batch Synchronization step type

- Batch Synchronization step settings
- Batch process model

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequences and steps in the example.

1. On the Sequences pane, select the **MainSequence**. The MainSequence contains four Sequence Call steps that correspond to the different approaches demonstrated in this example. When executed, the Batch process model executes four instances of this sequence, referred to as sockets 0 through 3.
2. On the Sequences pane, select the **Voltage Tests** sequence. This sequence uses the Batch Synchronization step settings to synchronize execution of individual steps. Observe that the two Apply Voltage Stimulus steps enable the **One thread only** option on the Synchronization pane of the step settings, indicating that only a single test socket execute these steps during execution. The two Voltage Response steps enable the **Parallel** option on the Synchronization pane of the step settings, indicating that all test sockets execute these steps concurrently.
3. On the Sequences pane, select the **Temperature Tests** sequence. This sequence uses the Batch Synchronization step type to define a block of steps that execute serially, meaning that only one socket executes each test at a time but that all sockets execute both test steps. Execution does not continue beyond the Batch Synchronization block until all sockets have completed the block.
4. On the Sequences pane, select the **Stress Tests** sequence. This sequence uses the Batch Synchronization step type to define blocks of steps that execute in parallel. For each block, all test sockets concurrently execute the test steps. Execution does not continue beyond the block until all sockets have completed all test steps within the block.
5. On the Sequences pane, select the **Current Tests** sequence. The steps in this sequence do not have any Batch Synchronization settings applied, meaning that

each test socket executes these steps as soon as the execution reaches this sequence. It is not necessary to specify Batch Synchronization settings for every step in a sequence file using the Batch process model.

Complete the following steps to run the example.

1. Select **Execute»Single Pass** to run the sequence.
2. As the test sequence executes, dialog boxes indicate the current status of the tests. Notice that the tests execute concurrently, serially, or only on a single socket according to the Batch Synchronization settings for each step.

Related concepts:

- [Batch Process Model](#)
- [TestStand Directory Structure](#)

Synchronization Step Types - Lock

Purpose

This example demonstrates how to use the Lock synchronization step to restrict execution flow in a sequence file that uses the Batch process model. When a lock is active, only one thread at a time can access the steps the lock contains.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\  
Synchronization Step Types\Synchronization Step Types -  
Lock.seq
```

Highlighted Features

- Batch process model
- Lock step type

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequence functionality.

1. Select **Edit»Sequence File Properties** to launch the Sequence File Properties dialog box. On the Advanced tab, the Model File control specifies the Batch process model, which will run the test sequence in multiple parallel executions (test sockets).
2. On the Sequences pane, select the **MainSequence**.
3. Review the steps in the `MainSequence`. The steps specify the following behaviors:
 - The sequence uses Lock steps to ensure that only one test socket is executing the single-threaded steps at a given time, which is useful in cases where the step requires a shared resource, such as an instrument, that only one step at a time can use.
 - In section one, the sequence first creates and then performs the lock operation in the Create and Lock "Lock 1" step. Once one test socket calls the Lock "Lock 1" step, all other test sockets wait at the step until the original socket releases the lock in the Unlock "Lock 1" step.
 - In section three, for the **Single-threaded Test 5** step, notice that the Synchronization panel of the Properties tab of the Step Settings pane contains a **Use Lock to Allow Only One Thread at a Time to Execute the Step** option, which is enabled. This option applies a lock for the step without the need for lock steps.
 - In section four, the **Create Lock** step stores the lock reference in a local variable. The lock operation uses this variable to identify the lock, and stores a reference to the lock operation in a second variable.

Complete the following steps to run this example.

1. Select the **Execute** menu and confirm that a checkmark appears next to the **Tracing Enabled** option.
2. Select **Execute»Single Pass** to run the sequence. TestStand creates three test sockets, and each test socket executes the `MainSequence`. The multi-threaded

steps execute simultaneously across all test sockets. Observe that the locked code runs only on a single socket at any given time.

Related concepts:

- [Batch Process Model](#)
- [TestStand Directory Structure](#)

Synchronization Step Types - Notification and Wait

Purpose

This example demonstrates how to use the Notification and Wait synchronization steps in a sequence file to specify that one test socket waits to receive a notification from another step before continuing the execution. A Notification step can send messages between multiple test sockets and can optionally contain data.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\  
Synchronization Step Types\Synchronization Step Types -  
Notification & Wait.seq
```

Highlighted Features

- Notification step type
- Wait step type

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequence functionality.

1. On the Sequences pane, select the **MainSequence**.
2. Complete the following steps to review the steps in the `MainSequence`.
 - a. Select the **Notification** step. On the Notification Settings edit tab, notice that the Create operation is selected. This step creates a new notification and names it "Event 1".
 - b. Select the **Wait for Notification** step. On the Notification Settings edit tab, notice that the Wait operation is selected. This step waits until TestStand calls the Set operation on the "Stimulus Started" notification from the "Start Stimulus" execution. On the Preconditions panel of the Properties tab of the Step Settings pane, the Precondition Expression control contains an expression that ensures that only Test Socket 1 waits for the notification. The other test socket skips this step.
 - c. Open the Start Stimulus sequence. Select the **Initialize Stimulus** step. The Wait Settings edit tab specifies that the Wait step introduces a 2-second delay in the sequence to simulate starting the stimulus.
 - d. Select the **Set "Stimulus Starting" Notification** step. On the Notification Settings edit tab, notice that the Set operation is selected. This step sends the notification. If another thread is waiting for the notification, that thread may now continue execution.

Complete the following steps to run this example.

1. Select **Execute»Single Pass** to run the sequence.
2. Notice that the main execution waits for the initialization to complete before entering the Main step group.
3. 2.3. Notice that the Wait for "Stimulus Started" Notification step does not complete until the "Start Stimulus" sequence sets the notification indicating that the stimulus has started.

Related concepts:

- [TestStand Directory Structure](#)

Synchronization Step Types - Queue

Purpose

This example demonstrates how to use the Queue synchronization step in a sequence

file that uses the Batch process model. You can use a Queue step to pass data between threads or across executions while avoiding race conditions.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\
Synchronization Step Types\Synchronization Step Types -
Queue.seq
```

Highlighted Features

- Batch process model
- Queue step type

Major API

None

Prerequisites

None

How to Use This Example

The Queue step is similar to a Notification step in that you can send data between multiple test sockets. Queue steps can contain multiple data values. This example demonstrates how to complete the following tasks for a queue:

- **Enqueue**—Add an element to a queue.
- **Dequeue**—Remove an element from a queue or store the data from the element.
- **Get Status**—Obtain information about the current state of the queue.
- **Flush**—Empty the queue and optionally obtain all the elements from the queue.

Complete the following steps to review the sequence functionality.

1. Select **Edit»Sequence File Properties** to launch the Sequence File Properties dialog box.
2. On the Sequences pane, select the **MainSequence**.
3. Complete the following steps to review the steps in the `MainSequence`.

- a. In section one, select the **Create Queue 1** step. On the Queue Settings edit tab, notice that the Create operation is selected. This step creates a new queue named "Queue 1".
- b. Select the Enqueue One Element sequence, then select the **Enqueue Data** step. On the Queue Settings edit tab, notice that the Enqueue operation is selected. This step adds a new element, the value of the `Locals.EnqueueData` variable, to the queue.
- c. Select the **Dequeue Data** step. On the Queue Settings edit tab, notice that the Dequeue operation is selected. This step removes an element from the queue. If the queue is empty, the step waits until data is available to dequeue, as shown in section two.
- d. In section three, select the **Get Status** step. This step uses the Get Status operation places the current contents of the queue into the `Locals.QueueStatus` variable without modifying the state of the queue.
- e. Select the **Flush Queue** step. On the Queue Settings edit tab, notice that the Flush operation is selected. This step removes all data from the queue. TestStand places the data in the location specified in the **Location to Store Array of Queue Elements** control. In this case, TestStand places the data in the `Locals.FlushedQueue` variable.

Complete the following steps to run this example.

1. Select **Execute»Single Pass** to run the sequence.
2. Read the text in each section popup message for information about the behavior of that section.

Related concepts:

- [Batch Process Model](#)
- [TestStand Directory Structure](#)
- [Synchronization Step Types - Notification and Wait](#)

Synchronization Step Types - Rendezvous and Semaphore

Purpose

This example demonstrates how to use the Rendezvous and Semaphore synchronization steps to control multi-threaded execution flow between threads or

across executions in a sequence file that uses the Batch process model.

- A Rendezvous step will make a set number of threads wait until all threads specified are at the same point and ready to proceed. Unlike batch synchronization, rendezvous steps can be used to synchronize any TestStand threads or executions, without requiring the batch process model.
- A Semaphore step is similar to the Lock step in functionality, except that you can specify any number of threads instead of just one.

Example File Location

```
<TestStand Public>\Examples\Built-In Step Types\
Synchronization Step Types\Synchronization Step Types -
Rendezvous & Semaphore.seq
```

Highlighted Features

- Batch process model
- Rendezvous step type
- Semaphore step type

Major API

None

Prerequisites

None

How to Use This Example

This example uses the Rendezvous or Semaphore steps to synchronize four test sockets in a batch test.

Complete the following steps to review the sequence functionality.

1. Select **Edit»Sequence File Properties** to launch the Sequence File Properties dialog box. On the Advanced tab, the Model File control specifies the Batch process model, which will run the test sequence in multiple parallel executions (test

sockets).

2. On the Sequences pane, select the **MainSequence**.
3. Complete the following steps to review the steps in the `MainSequence`.
 - a. Select the **Create Semaphore 1** step. On the Semaphore Settings edit tab, notice that the Create operation is selected. This step creates a new semaphore named "Semaphore 1". The Initial Semaphore Count control specifies a value of 2, which indicates that two test sockets can execute the semaphore section at any given time. The **Acquire Semaphore 1** step and Release Semaphore 1 step represent the start and end of the semaphore section.
 - b. Select the **Create Rendezvous 1** step. On the Rendezvous Settings edit tab, notice that the Create operation is selected. This step creates a new rendezvous named "Rendezvous 1". The Number of Threads per Rendezvous control specifies a value of 4, which indicates that when one test socket reaches the Start Rendezvous step, the socket waits for the other three sockets before continuing execution.

Complete the following steps to run this example.

1. Select **Execute»Single Pass** to run the sequence. TestStand creates four test sockets, and each test socket executes the `MainSequence`. When the executions reach the Acquire Semaphore 1 step, two test sockets enter the semaphore section and the remaining two sockets wait. When the first two test sockets execute the Release Semaphore 1 step, the remaining two test sockets enter the semaphore section. When the test sockets reach the Rendezvous step, the test sockets wait until all test sockets reach the step before execution continues.

Related concepts:

- [Batch Process Model](#)
- [TestStand Directory Structure](#)

Examples for Custom Step Types

The Custom Step Types examples extend TestStand with additional custom step types to call different types of test code. This directory includes the following example programs.

Agilent VEE Step Types

Purpose

This example uses ActiveX to call Agilent VEE code from TestStand using custom step types, which you can use to execute the user functions of VEE user libraries.

Example File Location

```
<TestStand Public>\Examples\Custom Step Types\Agilent VEE  
Step Types\Agilent VEE Computer Motherboard Test.seq
```

Highlighted Features

Custom step types

Major API

N/A

Prerequisites

VEE Evaluation, Development, or Run-Time Environment version 6.0 or later.



Note (64-bit TestStand) 64-bit TestStand does not support Agilent VEE step types.

How to Use This Example

The example sequence file, `Agilent VEE Computer Motherboard Test.seq`, contains a `SequenceFileLoad` callback that TestStand automatically executes when you open the sequence file and that performs the following necessary actions:

- Copies the TestStand example file `VEE_StepType.dll`, which implements the step type functionality, from the `<TestStand Public>\Examples\Custom Step Types\Agilent VEE Step Types` directory to the `<TestStand Public>\Components\StepTypes\Agilent VEE` directory and registers

the DLL.



Note `VEE_StepType.dll` was created using Microsoft Visual Basic. The example directory includes the `VEE_StepType.vbp` Visual Basic project file, which you can use to recompile `VEE_StepType.dll`.

- Copies the `<TestStand Public>\Examples\Custom Step Types\Agilent VEE Step Types\compMotherBoard.bmp` file to the default `Bitmaps` subdirectory of the VEE installation directory. The `SimulationDialog` VEE user function displays the bitmap on its view panel. When using a bitmap on a VEE view panel, you can reference the bitmap using an absolute path or you can place the bitmap in the `Bitmaps` subdirectory of the VEE installation directory.

When you open `<TestStand Public>\Examples\Custom Step Types\Agilent VEE Step Types\Agilent VEE Computer Motherboard Test.seq` in the TestStand Sequence Editor, the VEE step types appear in the VEE group on the Insertion Palette. You can make the VEE step types available for any sequence file by copying the VEE step types from `Computer.seq` in the Types window to a type palette, such as `MyTypes.ini`.

When you insert a VEE step type into a sequence, you must configure the step by clicking the **Specify VEE Module** button on the Step Settings pane to launch the Edit VEE Module dialog box. For the VEE Library to Load control, specify a valid VEE library. The step automatically populates the VEE Function ring control with user functions the VEE user library exports. When you select an exported function, the step automatically populates the input and output arguments of the function. The VEE step types currently support the following data types and data shapes for arguments of VEE user functions:

- Scalar:
 - Signed 32-bit integer
 - 64-bit real number (double)
 - String
- 1D-Array:
 - Signed 32-bit integer
 - 64-bit real number (double)

- String

You can select each input and output argument from the Argument control of the Input section and the Output section, respectively, and set a corresponding TestStand property path. The step uses the value of the property as input for input arguments or receives the output value for output arguments.

The VEE step types also support debugging. You can step into a VEE user function and use the VEE debug functionality to debug the user function.

This example includes VB_VEE_Action, VB_VEE_NumericLimitTest, VB_VEE_PassFailTest, and VB_VEE_StringValueTest step types. The VB_VEE_NumericLimitTest, VB_VEE_PassFailTest, and VB_VEE_StringValueTest step types combine the functionality of the VB_VEE_Action step with the built-in Numeric Limit Test, Pass Fail Test, and String Value Test step types, respectively, to call VEE functions in addition to the functionality of the built-in steps. For example, the VB_VEE_NumericLimitTest step returns a number from a VEE user function and compares the number to numeric limits.

Related concepts:

- [TestStand Directory Structure](#)
- [Custom Step Types](#)

ATLAS Step Type

Purpose

This example demonstrates a custom step type for calling an Abbreviated Test Language for All Systems (ATLAS) test program set (TPS), which is created in the PAWS Developer Studio. The ATLAS custom step type integrates with the PAWS Run-Time System (RTS) to execute TPS files (.PAWS). These examples require the TYX PAWS RTS Server to run the provided test program sets.

The ATLAS custom step type example includes the following example sequence files:

- A parameter passing example that demonstrates passing parameters between TestStand and a TPS.

- A manual intervention example that demonstrates support for handling a request for manual intervention by the PAWS RTS Server.

Example File Location

```
<TestStand Public>\Examples\Custom Step Types\ATLAS Step  
Types\Parameter Passing Example.seq
```

```
<TestStand Public>\Examples\Custom Step Types\ATLAS Step  
Types\Manual Intervention Example.seq
```

Highlighted Features

Custom step types

Major API

N/A

Prerequisites

TYX PAWS RTS 1.22 or later

How to Use This Example

The example sequence files contain a `SequenceFileLoad` callback that TestStand automatically executes when you open either sequence file. The `SequenceFileLoad` callback copies the `ATLAS_StepType.dll` file from the `<TestStand Public>\Examples\Custom Step Types\ATLAS Step Types` directory to the `<TestStand Public>\Components\StepTypes\ATLAS` directory and registers the DLL. The ATLAS custom step type calls functions within `ATLAS_StepType.dll` to configure and handle interactions between TestStand and PAWS.

You must install the PAWS RTS with the TPS Server before you can build the accompanying project. You must contact TYX Corporation for more information about obtaining the PAWS RTS.

The project assumes you have installed the PAWS RTS in the default location of

C:\usr. If you installed the PAWS RTS in a different location, modify the path in the following lines of code, located in <TestStand Public>\Examples\Custom Step Types\ATLAS Step Types\ATLAS\StdAfx.h, to reflect the new location:

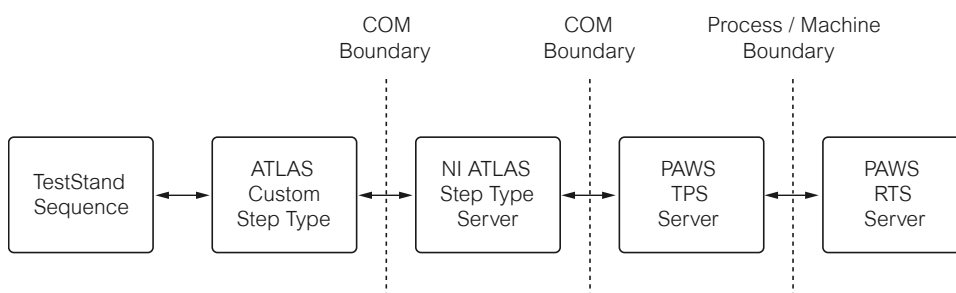
- Change the line `#import "C:\usr\tyx\com\rtsax.dll"` to `#import "<New Location>\tyx\com\rtsax.dll"` raw_interfaces_only
- Change the line `#import "C:\usr\tyx\com\ComUtil.dll"` to `#import "<New Location>\tyx\com\ComUtil.dll"` raw_interfaces_only

When you set the paths correctly, you can build the ATLAS custom step type project with Microsoft Visual C++ 6.0.

The ATLAS custom step type creates the PAWS RTS and executes a TPS through the PAWS TPS COM Server, and communicates with the TPS server by calling methods within the NI ATLAS Step Type ActiveX server located in `ATLAS_StepType.dll`. The server contains the `IExecuteTPS` class, which includes the following methods:

- **Edit Step**—Sets up a TPS to call and the parameter to pass to the TPS.
- **Execute TPS**—Creates the PAWS RTS and executes a TPS.

The ATLAS custom step type supports the use of the TestStand numeric, Boolean, and string types. You can use parameters of these types to pass data between TestStand and the TPS server. Data is exchanged through COM boundaries between the ATLAS custom step type in TestStand, the NI ATLAS Step Type server, and the TPS server, which monitors the PAWS RTS server, as shown in the following figure.



Parameter Passing Example

When you open the `ParameterPassingExample.seq` example sequence file, the `SequenceFileLoad` callback copies the required files to the appropriate directories. When TestStand executes the Main step group of the MainSequence, TestStand launches a dialog box that requests a numeric value to pass to the sample TPS. TestStand passes the numeric value, along with various other parameters, the TPS server. The sample TPS modifies the numeric value and parameters and shows the modified values when execution returns to TestStand.

Manual Intervention Example

When you open the `ManualInterventionExample.seq` example sequence file, the `SequenceFileLoad` callback copies the required files to the appropriate directories. When TestStand executes the Main step group of the MainSequence, the ATLAS step calls a TPS that simulates and tests several devices. When the TPS simulates the DMM, TestStand launches a dialog box that prompts for the simulated voltage. If you enter a voltage that is within the acceptable range of 21 to 23 V, the test passes. Otherwise, the test fails.

During execution, the TPS server also simulates a manual intervention request, in which you can pause the execution while the TPS server resolves the condition that caused the request. The simulated manual intervention launches a dialog box on the same computer on which TestStand is running to inform you that manual intervention has been requested. Click **Continue** in the dialog box to resume TPS execution from TestStand.

Using the ATLAS Step in New Sequence Files

You can call an ATLAS TPS using step instances of the ATLAS custom step type. The example sequence files contain ATLAS steps preconfigured to call a sample TPS. You can execute these sequences without making changes. You can also use the ATLAS custom step type in your own sequence files to call an ATLAS TPS. You can insert an ATLAS step into a sequence in one of the following ways:

- Copy an ATLAS step from one of the example sequence files to the new sequence.
- Copy the ATLAS custom step type from one of the example sequence files to the new sequence.

- Add the ATLAS custom step type to a type palette file and use the Insert Step submenu in the Steps Pane context menu to create an instance of an ATLAS step.

Before you can execute a sequence that contains an ATLAS step, you must configure the step to call the appropriate TPS and pass any desired parameters. Click the **Edit ATLAS Step** button on the Step Settings pane to launch the Edit ATLAS Call dialog box, in which you can specify the TPS the step executes and the parameters passed between TestStand and the TPS.

You can then execute the sequence that contains the steps. The ATLAS custom step type fails on an ATLAS step if any faults are reported when the TPS executes. The ATLAS custom step type also propagates and reports any errors the TPS server encounters to TestStand. If the TPS server encounters no errors or faults, the step passes.

When TestStand first executes an ATLAS step, the ATLAS custom step type loads an instance of the RTS, or attaches to an existing RTS if one already exists. TestStand stores a reference to this instance as a temporary global variable in the TestStand Engine. Future executions of the ATLAS custom step type use the same reference for TPS executions. The engine releases the reference when it shuts down.



Note TestStand updates properties referred in the Value Expression control in the Edit ATLAS Call dialog box with the parameter values the TPS server passes after TestStand calls the TPS server when executing an ATLAS step.

Limitations

The following are known limitations of the ATLAS custom step type:

- The step type does not support passing TestStand arrays, containers, or object references as parameters.
- You can execute only one ATLAS step at a time for each local or remote computer. The ATLAS steps are locked globally with respect to the name of the local or remote computer. A sequence might contain multiple ATLAS steps that pertain to the same local or remote computer, but the step executions are sequential. You can execute ATLAS steps that pertain to different computers simultaneously.
- The TestStand Deployment Utility does not automatically copy and register ActiveX

servers. You must add `ATLAS_StepType.dll` to the list of deployment files to deploy the ATLAS custom step type. When you deploy the step type, you must register the NI ATLAS Step Type ActiveX server. Refer to the `SequenceFileLoad` callback sequence in either ATLAS example sequence file for an example of registering an ActiveX server.

Error Codes and Descriptions

You might receive some error codes when using this example.

Related concepts:

- [TestStand Directory Structure](#)
- [Custom Step Types](#)
- [Type Palette Files](#)
- [Edit ATLAS Call Dialog Box](#)
- [Deploying TestStand Systems](#)
- [Error Codes and Descriptions](#)

Edit ATLAS Call Dialog Box

Select an ATLAS step in a sequence and click the **Edit ATLAS Step** button on the Step Settings pane to launch the Edit ATLAS Call dialog box, in which you can specify the test program set (TPS) the step executes and the parameters passed between TestStand and the TPS.

The Edit ATLAS Call dialog box contains the following options:

- **TPS**—The file path to the TPS the step executes. The control below the TPS control shows the complete file path of the selected TPS and indicates whether the file was found.
- **Run TPS on a Remote Computer**—When enabled, the TPS executes on a remote computer. When disabled, the TPS executes on the local computer. This option is disabled by default.
 - **Remote Host Name Expression**—The name of the remote computer. This option is available only when you enable the Run TPS on a Remote Computer option. You can enter the name in the control or click the **Network Browser Dialog Box** button to launch the Browse for Computer dialog box and select

the remote computer. If you enter the name in the control, you must use quotes around the name because the content of the control is a TestStand expression. You can specify a TestStand expression instead of a hard-coded name. TestStand evaluates the expression at run time to resolve the name of the remote computer. You can click the **Expression Browser Dialog Box** button to launch the Expression Browser dialog box, in which you can construct the expression.

- **Parameters**—The parameters passed between TestStand and the ATLAS TPS. TestStand parameters correspond to PAWS global variables, parameters, and results. You must specify parameters to pass only when you pass data between TestStand and the TPS.
- **Add Parameter**—Adds a new parameter to the list of parameters to pass. Use the Parameter Details section to configure the new parameter.
- **Delete Parameter**—Deletes the selected parameter from the list of parameters.
- **Move Parameter Up**—Moves the selected parameter up in the list of parameters.
- **Move Parameter Down**—Moves the selected parameter down in the list of parameters.
- **Parameter Details**—Specifies the following information for the selected parameter in the list of parameters:
 - **Name**—The name of the parameter.
 - **Type**—The data type for the number. The type can be a number, string, or Boolean.
 - **Mode**—Specifies if the data type mode is read, write, or read and write.
 - **Global Variable**—When enabled, the parameter is a PAWS global variable or a PAWS parameter/result.
 - **Value Expression**—The value expression for the parameter. The sequence context used to evaluate parameter values is the step context.



Note

- If you can specify the file path to the TPS using relative or absolute paths, TestStand launches a File Not Found dialog box that contains options for specifying the file location.
- You must properly configure the DCOM settings for the PAWS Run-Time Server (RTS) on both the host and remote computers for remote execution to work. Refer to PAWS documentation for more information

about configuring the PAWS RTS for remote execution.

- You must specify the name of the parameter value exactly as the TPS and TPS server expect. The TPS server allows verification of parameter names prior to execution. The ATLAS custom step type generates an error if the parameter cannot be read or written.
- TestStand updates properties referred in the Value Expression control with the parameter values the TPS server passes after TestStand calls the TPS server when executing an ATLAS step.

Related concepts:

- [ATLAS Step Type](#)
- [Sequence Context](#)

Error Codes and Descriptions

The following table lists possible error codes you might receive while using the ATLAS step type example and the corresponding descriptions.

Error Code	Description
-17001	The test program set (TPS) cannot continue execution because the NI ATLAS Step Type ActiveX server is in an unrecognized state. Unload the server from TestStand and restart the execution.
-17205	The TPS cannot be executed because the PAWS Run-Time System (RTS) is in a bad state or cannot be initialized. Terminate the current instance of the PAWS RTS and run the TPS again.
-17208	The specified TPS file was not found. Ensure that the file path is correctly specified so TestStand can locate the file and its directory.
-17401	The TPS cannot be executed because another TPS exists and is currently running on the requested host. Only one instance of the PAWS RTS can exist on a host at any time. Wait for the currently running TPS to complete execution or terminate the execution before executing another TPS.
-17500	The execution failed because the RTS reported an error. The RTS error displays as part of the error message.

Error Code	Description
-17501	An unexpected error occurred.
-17600	The TPS cannot be loaded because the RTS was interrupted during the load process. Check the TPS and PAWS RTS for errors and try running the TPS again.
-17601	No TPS file was specified for execution. Use the Edit ATLAS Call dialog box to select a file to call before executing the step.

Related concepts:

- [ATLAS Step Type](#)
- [Edit ATLAS Call Dialog Box](#)

Java Step Types

Purpose

This example demonstrates custom step types for calling methods in Java classes. The step types call compiled Java class files (.class) using the Java Native Interface (JNI) and permit the invocation of methods located within the classes.

Example File Location

```
<TestStand Public>\Examples\Custom Step Types\Java Step  
Types\Java Computer Motherboard Test.seq
```

Highlighted Features

Custom step types

Major API

Engine.UnloadAllModules

Prerequisites

Java Runtime Environment (JRE) version 6 update 16, which you can download from

the Oracle website. The Java step types are likely compatible with earlier and later versions of the JRE but have not been tested in JRE versions other than version 6 update 16.

How to Use This Example

The example sequence file, `Computer.seq`, contains a `SequenceFileLoad` callback that TestStand automatically executes when you open the sequence file. The `SequenceFileLoad` callback copies the `JavaCall.dll` and `JavaCall.uir` files from the `<TestStand Public>\Examples\Custom Step Types\Java Step Types` directory to the `<TestStand Public>\Components\StepTypes\Java` directory. The Java step types call functions within `JavaCall.dll`, which must be located in a TestStand search directory for the step types to locate the DLL.



Note You must install the JRE before you can build the accompanying project.

The Java step types support the creation and cleanup of the Java virtual machine (JVM) and the manipulation of objects that exist within the JVM through method calls. The step types support the use of all Java primitive types and strings, which you can use to pass data between TestStand and Java objects. The step types rely on `JavaCall.dll`, which is a LabWindows/CVI library, for TestStand to interface with Java. The library communicates with TestStand using an ActiveX interface and communicates with Java through the JNI.

Java uses the JNI to call compiled code written in C/C++ and to access Java code from C/C++. Refer to the Java Native Interface website for more information about JNI technology.

When you open `<TestStand Public>\Examples\Custom Step Types\Java Step Types\Java Computer Motherboard Test.seq` in the TestStand Sequence Editor, the Java step types appear in the following groups on the Insertion Palette:

- **StartStop**—Contains steps that start and destroy the JVM.
- **Methods**—Contains steps that call a particular Java method in a specified class.

- **Static Methods**—Contains steps that call a particular static class method in a specified class.

The CallStaticJavaMethod and CallJavaMethod steps consist of Numeric Limit Test, String Value Test, Pass Fail Test, and Action steps.

Use a Java step type to call `JavaCall.dll`, which extracts information about the step type from TestStand using ActiveX calls and then calls the appropriate method in the JVM using the JNI. For example, use the CallJavaMethod step to call the InvokeJavaMethod function `JavaCall.dll`, which calls the TestStand Engine to extract the class, method, and parameter information and instantiate the class within the existing JVM. The step identifies and calls the method with parameter values retrieved from TestStand. The `JavaCall.dll` file then passes the return value back to TestStand.

StartStop Steps

Use the StartJVM and DestroyJVM steps, located under the StartStop group on the Insertion Palette, in pairs around other Java steps to call methods for the contained steps. A typical sequence structure begins with a StartJVM step, which you place in the Setup step group, followed by a number of Methods or Static Methods steps, which you place in the Main step group. Place the DestroyJVM step in the Cleanup step group.

Configure the StartJVM step to provide the class path, which is the path of the Java classes to access. Select the StartJVM step in a sequence and click the **Edit Java Class Path Location** button on the Step Settings pane to launch the Set Class Path dialog box, in which you can specify the class path. You must include all Java classes that are accessed in the path. If you leave the path empty, the JVM cannot locate the user-written Java classes. Click **Browse** in the Set Class Path dialog box to launch the Select Paths dialog box, in which you can add directories that contain the Java classes to call.

Methods and Static Methods Steps

The Static Methods steps call static class methods and the Methods steps call other methods of the class. The Methods steps automatically load the appropriate class and create new instances of the class before making the call to the method. The Static Methods steps do not create new instances of the class before making the call.

Methods and Static Methods steps include the following steps:

- **Numeric Limit Test**—Used to call methods that return a numeric value, such as a byte, integer, short, long, float, or double. You can compare the value returned to a set of limits to determine whether the step passed.
- **String Value Test**—Used with a method that returns a string value. The step compares the returned value to the expected value.
- **Pass/Fail Test**—Used with Boolean return values.
- **Action**—Used with any type of return value.

You can configure Methods and Static Methods steps by selecting a step in the sequence and clicking the **Edit Call** or **Edit Static Call** button on the Step Settings pane to launch the Edit Java Call dialog box, in which you can configure the step to call the appropriate Java method.

Tips for Using the Java Step Types

Keep the following tips in mind when you use Java step types in TestStand:

- Before calling any Java methods, you must create the JVM. Call the StartJVM step before you call any of the Methods or Static Methods steps.
- The Methods steps automatically load the appropriate class and allocate an object of that type. The Static Methods steps load only the appropriate class. Neither step type automatically calls the constructor of the class. The example does contain a sample .
- Because Java is case-sensitive, the class name and method names must use the same case as the Java class file specifies.

Compatible Types

The following table shows how data types in TestStand are compatible with certain data types in Java. The Java step types automatically handle the conversion between types.

TestStand Type	Java Types
Number	byte, short, int, long, float, double
String	char, string

TestStand Type	Java Types
Boolean	Boolean

Computer Motherboard Example

When you open the `Computer.seq` example sequence file, the `SequenceFileLoad` callback copies the required files to the appropriate directories. When TestStand executes the Setup step group of the MainSequence, the sequence first obtains the TestStand directory and locates the Java classes to call. The `StartJVM` step automatically sets the class path and passes it to the JVM. TestStand then calls a static Java method called `Start`, which creates a new computer object by calling the constructor of the `Computer` class, to launch a dialog box. You can dismiss the dialog box, and TestStand reads the tests you select to fail from the Java class one call at a time because Java supports only calls by value semantics.

When the calls are complete, the steps in the Main step group call the Java methods. The steps supply a value to the methods that determine whether the test passes or fails. Depending on the supplied value, the method returns a result. If a single step fails, TestStand skips all remaining steps.

The step in the Cleanup step group destroys the JVM.

Limitations

The following are the known limitations of the Java step types:

- The step types currently do not support Java reference types, such as arrays and objects.
- You cannot access the same instantiated object multiple times. Each time a step calls a method on an object, a new object is created. Therefore, all states must be maintained in class variables as static members.
- The step types cannot directly call constructors. The example does contain a sample .
- The step types are not thread-safe. Do not use the step types with the Parallel and Batch process models.

Known Issues

Because you cannot unload the JVM once you load it in a process without exiting the process, the DestroyJVM step does not actually destroy the JVM. Instead, the DestroyJVM step only detaches the current thread, which indicates that the JVM is no longer referenced, so that the JVM can unload in future versions, if required. Refer to the DestroyJavaVM API documentation on the Oracle website for more information about this issue.

You cannot unload the JVM once you load it in a process, and you cannot unload `JavaCall.dll` when you call the `Engine.UnloadAllModules` method of the TestStand API. You can unload `JavaCall.dll` only by exiting the process by closing the sequence editor or user interface. Once you load a Java class file, you must restart the process by restarting the sequence editor or user interface to force the JVM to load any changed or recompiled versions of the Java class.

You cannot change the class search path once TestStand loads the JVM. When TestStand calls the StartJVM step with a given class path, you cannot change the class search path by calling the StartJVM step again with a new class path. You can change the class path only by exiting the process by closing the sequence editor or user interface. You must set the required class paths in the first StartJVM step.

Error Codes and Descriptions

You might receive some error codes when using this example.

Related concepts:

- [TestStand Directory Structure](#)
- [Custom Step Types](#)
- [Edit Java Call Dialog Box](#)
- [Error Codes and Descriptions](#)

Edit Java Call Dialog Box

Select a Java Methods and Static Methods step in a sequence and click the **Edit Call** or **Edit Static Call** button on the Step Settings pane to launch the Edit Java Call dialog box, in which you can configure the step to call the appropriate Java method.

The Edit Java Call dialog box contains the following options:

- **Java Class to Load**—The Java class that contains the method to call. If TestStand cannot locate the file, TestStand launches the File Class Not Found dialog box. Select the **Add the directory containing the file you selected to the list of search directories** option and click **OK**. Do not select **Use an absolute path for the file you selected**. The control below the Java Class to Load control displays the complete file path of the specified class. If the file is incorrect, add the appropriate directory to the list of TestStand search directory paths or move the directory up to a higher position in the list of search paths.



Note Always use a relative path for the Java class to load. If you specify an absolute path, the Java virtual machine (JVM) cannot locate the class name and you receive a -1000 Class Not Found error.

- **Java Method**—The name of the method to call. Specify the name after you select the Java class that contains the method.
- **Parameters**—Contains the following options for creating and configuring parameters to pass to the specified method:
 - **Method Parameter**—The list of parameters to pass to the specified method. Select a parameter from the ring control to specify the argument type and value expression for the parameter. The order of parameters must match the order the Java method header uses in the parameter list. If no parameters are to be passed, delete all parameters from the list.
 - **New**—Adds a new parameter to the list of parameters.
 - **Argument Type**—The type of the parameter. The available types are the Java types represented in the Java method header.
 - **Delete**—Deletes the selected parameter from the list of parameters.
 - **Value Expression**—The value expression for the parameter. If the value to be passed is stored in a TestStand variable, click **Browse** to locate the variable. Otherwise, enter the value of the parameter. You must verify that the type of the value or TestStand variable is compatible with the Java parameter type.



Note Enclose literal strings and characters in double quotes.

- **Return Value**—Contains the following options for specifying a return value for the specified method:
 - **Return Value**—The return value.

- **Argument Type**—The type of the return value. The type must match the variable that stores the return value, or the return value must be compatible for the appropriate comparisons to occur. For example, the Numeric Limit Test steps require the return value to be stored in `Step.Result.Numeric`, the String Value Test steps require the return value to be stored in `Step.Result.String`, and the Pass Fail Test steps require the return value to be stored in `Step.Result.PassFail`. For Action steps, select **void** for the argument type if no return value exists.
- **Value Expression**—The value expression for the return value, which TestStand stores in a variable. Click **Browse** to locate the variable. You must verify that the type of the value or TestStand variable is compatible with Java types.

Related concepts:

- [Java Step Types](#)
- [Search Directories](#)

Error Codes and Descriptions

The following table lists possible error codes you might receive while using the Java step types example and the corresponding descriptions.

Error Code	Error Message	Explanation
-1001	Could not launch JVM	The creation of the Java virtual machine (JVM) failed. Refer to the Sun website at <code>java.sun.com</code> for information about the possible causes of the failure. Verify that <code>jvm.dll</code> was found and is not corrupt, and ensure that the class path is valid.
-1001	Could not attach thread to JVM	The current thread was detached from the JVM. Restart TestStand and ensure that the Java run-time environment is installed correctly.
-1001	Could not detach current thread	The current thread could not be detached from the JVM. Restart TestStand and ensure that the Java run-time environment is installed correctly.
-1002	Could not launch JVM	The current thread was detached from the JVM. Restart TestStand and ensure that the Java run-time environment is installed correctly.

Error Code	Error Message	Explanation
-1003	Could not launch JVM	A Java Native Interface (JNI) version error occurred. The wrong version of the JVM was found. Ensure that the Java run-time environment is installed correctly.
-1004	Could not launch JVM	Insufficient memory to create the JVM.
-1005	Could not launch JVM	The JVM already exists in memory.
-1006	Could not launch JVM	Invalid arguments are passed. Verify that the class path is correct and ensure that the Java run-time environment is installed correctly.
-1500	Cannot find class	The specified class name was not found in the class path. Verify the class name and that the file that contains the class exists in the class path.
-2000	Incorrect class name	The specified class name is incorrect. Re-enter the class name in the Edit Java Call dialog box.
-3000	Cannot find method	The specified method was not found in the specified class. Verify that the method name is correct and that it exists in the specified class.
-3500	Too many arguments passed for void method	Too many arguments are passed for a void method.
-4000	Error reading returned string	The string returned from Java is corrupt. Verify that the return type of the Java method is a string value.
-5000	Invalid return type found	The return type of the Java method does not match the return type specified in TestStand. Ensure that the variable types in TestStand are compatible with the Java types.
-6000	JVM not invoked	The JVM was not created. Verify that TestStand calls the StartJVM step before this step.
-7000	Unable to find registry key for Java run time	The registry key that contains the entry for the Java run-time environment cannot be found. The JVM installer registers the key. Reinstall the JVM.
-7500	Unable to find registry keys for Java	The registry entry that specifies the location of <code>jvm.dll</code> was not found. The Java run-time environment installs the entry. Reinstall the JVM.

Error Code	Error Message	Explanation
-8000	Unable to find run-time DLL for Java	The registry entry for <code>jvm.dll</code> points to an invalid location, which indicates that the DLL was not found at the location. Reinstall the JVM.
-9000	Unable to find CreateJavaVM function jvm.dll	The <code>jvm.dll</code> file does not contain the <code>CreateJavaVM</code> function, which indicates that the DLL is invalid. Reinstall the JVM.
-10000	Cannot allocate memory	The system is low on resources and memory allocation failed.

Related concepts:

- [Java Step Types](#)
- [Edit Java Call Dialog Box](#)

Examples for Customizing Result Processing

The Customizing Result Processing examples demonstrate common approaches for customizing default TestStand result processing plug-ins, and for creating custom result processing plug-ins. This directory includes the following example programs.

Adding Custom Data to a Report

Purpose

You must complete the following tasks to add additional data to a report:

- Configure the desired property to be logged as a result. Results are stored in the `Locals.ResultList` array.
- Configure the result to be added to the report. This is typically accomplished by enabling the `\IncludeInReport\` flag.

This example demonstrates the following approaches for adding custom data to a

report:

- Using the UUT.AdditionalData container to add data to the report header
- Using the API method Execution.AddExtraResult
- Creating a custom property and setting the IncludeInReport flag
- Creating a custom step type with custom properties in the Step.Result container
- Using the Additional Results step type or property setting

Example File Location

```
<TestStand Public>\Examples\Customizing Result Processing\  
Adding Custom Data To a Report\Adding Custom Data To a  
Report.seq
```

Highlighted Features

- TestStand API
- PreUUT Callback
- Custom step types

Major API

- Execution.AddExtraResult
- PropertyObject.SetValString
- PropertyObject.SetFlags

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to execute the sequence.
2. As the example executes, read the message popups for information about each method demonstrated.

Complete the following steps to review the sequences and steps in this example.

1. Select the **PreUUT Callback** in the sequences pane. Note that the statement steps create new properties in the `Parameters.UUT.AdditionalData` container, which will appear in the report header. The `PropertyObject.SetFlags` method is used to configure the property to display in the report.
2. Select the **AddExtraResult** sequence. This sequence calls the `Execution.AddExtraResult` method to add the `Step.MessageExpr` property to the Result list. For the property to appear in the report, the `IncludeInReport` flag must be enabled for this property.
3. Select the **CustomProperty** sequence. This sequence uses the `PropertyObject.SetValString` method to create a new property in a Numeric Limit Test step. The sequence then calls `PropertyObject.SetFlags` to enable the `IncludeInReport` flag for the new property so that it is displayed in the report.
4. Select the **CustomStepType** sequence. This sequence calls a step type which has an additional property in the results container. The `IncludeInReport` flag for this property is enabled in the step type definition, so you do not need to enable it for each instance of the type.
5. Select the **AdditionalResults** sequence. This sequence uses the Additional Results step to log additional data to the report. It also uses the Log option for parameters in the Calculate Area of Circle code module to log the parameter values

Related concepts:

- [Storing Additional Information for UUTs and Test Stations in Process Models](#)
- [TestStand Directory Structure](#)
- [Custom Step Types](#)

Adding Custom Images to a Report

Purpose

This example is identical to the `computer.seq` example included in the `<TestStand Public>\Examples\Demo` directory except that it demonstrates how to modify the report of any failed diagnostic test steps to include HTML hyperlinks. When you are using an HTML-compatible report viewer, scroll down in the report to view any failed tests.



Note You can enable report generation in the ReportOptions callback.

Example File Location

```
<TestStand Public>\Examples\Customizing Result Processing\  
Adding Custom Images To a Report\Adding Custom Images To a  
Report.seq
```

Highlighted Features

Generating a test report

Major API

None

Prerequisites

None

How to Use This Example



Note Before you run this example, click the **Toggle Analyze File Before Executing** button on the Sequence Analyzer toolbar to specify that you do not want the TestStand Sequence Analyzer to analyze the sequence file before TestStand executes the sequence. If you leave sequence analysis enabled and you run the sequence, TestStand returns an error because the sequence file generates some variables dynamically at run time. If you do encounter this error, you can select the **Continue Execution** option to continue sequence execution.

Complete the following steps to use this example.

1. Select **Execute»Single Pass** to run the sequence. TestStand launches the Test Simulator dialog box, in which you select a computer motherboard test that you want to fail.
2. Select at least one test to fail and click **Done**. After execution completes, TestStand

displays the report on the Report pane.

3. Review the report. The Failure Section of the report lists the steps that failed and, for each failed step, includes an HTML hyperlink that links to the section of the report with more information about the step.
4. Close the Report pane.
5. In the Sequence File window, select the second If step. The If Condition edit tab specifies an expression that determines whether any test in the sequence failed. If a test failed, the sequence checks the status of each test and calls `computer.dll`, located in the `<TestStand Public>\Examples\Customizing Result Processing\Adding Custom Images To a Report\HTMLDiagnosticLinks` directory, to determine what information to add to the report. TestStand passes value by reference so the DLL can update the report.

This example uses a DLL to maintain modular and reusable code, but you can also configure each step that fails to add HTML content directly to the report.

Related concepts:

- [TestStand Directory Structure](#)

Displaying Graphs in a Report

Description

This example demonstrates how to include an array from a step that calls a code module in the test report.

Example File Location

```
<TestStand Public>\Examples\Customizing Result Processing\
Displaying Graphs In a Report\Displaying Graphs In a
Report.seq
```

Highlighted Features

Custom step types

Major API

None

Prerequisites

None

How to Use This Example

To execute the sequence, select **Execute»Single pass**.

The **Methods of Logging Array Data** section of the MainSequence demonstrates the following two approaches for adding array data to the report:

- **Log Array using Custom Action Step** is an instance of a custom step type that specifies the custom property `Step.Result.ResultArray`. The Custom Action Step calls a code module that generates an array and writes it to `Step.Result.ResultArray`. TestStand step types set the `IncludeInReport` property flag on a subset of their result properties to specify that these properties automatically appear in the report. The `IncludeInReport` flag has been enabled for the `Step.Result.ResultArray` property so that TestStand automatically adds the values of `ResultArray` to the report during report generation.
- **Log Array using Additional Results** uses the Log field to specify that a parameter should be in the report. This field is configured in the Parameters pane, located in the module tab of the step settings pane. The log field allows you to easily log data with no further customization.

You can customize how TestStand displays the array in the report with the **Include Arrays** option on the Contents tab of the Report Options dialog box. TestStand can insert arrays as tables or graphs. In this example, the ReportOptions callback sets arrays to display as graphs.

In the **Using Graph Attributes** section of the example, you can customize the graph of an array by using the attributes of the array property, `Locals.OutputData`. To view or edit the attributes of a property, right click the property, then select **Advanced»Edit Attributes**. To customize the appearance of a graph, use the

`DataLayout` and `DataOrientation` attributes, as described in the [Displaying Array Data as Graphs](#) help topic.

Review the comments for the **Set Graph Attributes and Log** steps for more information about the graph behavior for each value of the `DataLayout` and `DataOrientation` attributes.

Related concepts:

- [TestStand Directory Structure](#)
- [Custom Step Types](#)

Including Hyperlinks in a Report

Purpose

This example demonstrates how to include hyperlinks in reports that TestStand generates. You can use the Path type, with the attribute `TestStand.Hyperlink` set to `True`, to convert any path displayed in the report to a hyperlink.

Example File Location

```
<TestStand Public>\Examples\Customizing Result Processing\  
Including Hyperlinks in a Report\Including Hyperlinks in a  
Report.seq
```

Highlighted Features

Path data type

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to run the example.

1. To link to an image, select the **Create a link to an image on disk** step. This step stores an image path in a local variable and adds this variable to the report as an additional result. This data must be stored in a TestStand property before logging so that the hyperlink attribute can be set to `True` for the property. Click **Edit Attributes** next to the local variables to view this attribute.
2. To link to a web page, select the **Create a link to a webpage** step. This step stores a URL in a local variable and adds this variable to the report as an additional result.
3. Select the **CPU Test** step. If this step fails, an additional result is logged to link to an HTML file that contains troubleshooting instructions for the operator.
4. Select **Execute»Single Pass**.
5. When the execution completes, review the report, which contains links to the image, web page, and local HTML file.
6. Click the links to open the corresponding locations.

Related concepts:

- [TestStand Directory Structure](#)
- [Path](#)

Including Hyperlinks in a Report - TDMS File

Purpose

This example demonstrates how to use the `NI_TDMSReference` and `Path` data types to include hyperlinks to files in reports TestStand generates.

Example File Location

```
<TestStand Public>\Examples\Customizing Result Processing\  
Including Hyperlinks in a Report\Including Hyperlinks in a  
Report - TDMS File.seq
```

Highlighted Features

- NI_TDMSReference data type
- Path data type

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to use this example.

1. Select the **Provide links to TDMS file and Image in the generated report** step. This step adds the `TDMSRef` and `ImagePath` variables to the `ResultList` using the Additional Results feature.
2. Select **Execute»Single Pass**.
3. When the execution completes, review the report, which contains links to the TDMS file and the image. Text reports do not include hyperlinks.
4. Click the links to open the corresponding files. TestStand uses the default application you specify for the file format to open the file when you click the link in the report.

Related concepts:

- [NI_TDMSReference](#)
- [Path](#)
- [TestStand Directory Structure](#)

Model Plug-in - Basic Step Time Report

Purpose

This example demonstrates how to a basic step time report for each UUT so you can benchmark and analyze the performance of a sequence file. The example generates a

summary report, a file of unprocessed data, and a report in format that includes processed time data. You can also run a stand-alone executable to a basic step time report in `.csv` format to Excel format. You can basic step time reports in Excel format for multiple UUTs.

Example File Location

```
<TestStand Public>\Examples\Customizing Result Processing\
ModelPlugin Basic Step Time Report\Install Basic Step Time
Report Plugin.seq
```

Highlighted Features

Process model plug-in architecture

Major API

None

Prerequisites

You must have Microsoft Office Excel 2007 SP2 or later installed on the computer to a basic step time report in Excel format and to run the executable to a `.csv` file to Excel format. If Excel is not installed on the computer, this custom reporting plug-in generates only a `.csv` file.

How to Use This Example

You can use this example to a basic step time report. The example generates a summary report, a file of unprocessed data, and a report in format that includes processed time data. You can also run a stand-alone executable to a basic step time report in `.csv` format to Excel format. You can basic step time reports in Excel format for multiple UUTs.

Generating a Basic Step Time Report

Complete the following steps to use this example to generate a basic step time report.

1. Open `<TestStand Public>\Examples\Customizing Result`

Processing\ModelPlugin Basic Step Time Report\
 SetupBasicStepTimeReportGenerator.seq, click **OK** in the About This
 Example dialog box to copy the example and support files to the <TestStand
 Public>\Components\Models\ModelPlugins directory, and click **Yes** in
 the Setup Simple Text Report Example Files dialog box.

2. Open a sequence file for which you want to generate a basic step time report.
3. Select **Configure»Result Processing** to launch the Result Processing dialog box.
4. Select **Basic Step Time Report** from the context menu to add the Basic Step Time Report plug-in to the active configuration.
5. Enable the checkbox in the Display column of the Basic Step Time Report plug-in to display the HTML summary report in the Report pane.
6. Click the icon in the Options column of the Basic Step Time Report plug-in to launch the Basic Step Time Report Configuration dialog box, in which you specify the directory in which to store the report and specify whether to launch the report in Excel.
7. Click **OK** in the Basic Step Time Report Configuration dialog box and in the Result Processing dialog box to return to the Sequence File window.
8. Execute the sequence file and view the reports.

Convert a .csv File to Excel Format

Complete the following steps to use a stand-alone application to convert a .csv file to Excel format.

1. Launch <TestStand Public>\Examples\Customizing Result Processing\ModelPlugin Basic Step Time Report\CsvToBasicStepTimeReportGenerator.exe.
2. Specify the path to the .csv file you want to convert in the **Path of CSV Report** control and specify the path to the Excel file you want to create in the **Path of Basic Step Time Report** control.
3. Enable the **Launch Basic Step Time Report to open the report in Excel** option and click **OK**.
4. View the report in Excel.

You can also use command-line arguments to launch the stand-alone application, specify the path to the .csv file, specify the path of the Excel file you want to create, and optionally display the file in Excel as follows:

```
CsvToBasicStepTimeReportGenerator.exe <Path of CSV Report>
<Path of Excel File to Create> [/display]
```

Aggregating Basic Step Time Reports in Excel Format for Multiple UUTs

Complete the following steps to aggregate basic step time reports in Excel for multiple UUTs. The value in the ID column of the Raw Data worksheet must be unique across all the reports you want to aggregate.

1. Open the reports you want to combine in Excel.
2. In the main report file into which you want to aggregate all the data, unprotect the **Raw Data** worksheet.
3. Complete the following steps for each report file you want to aggregate into the main report file.
 - a. From the **Raw Data** worksheet, copy the rows that correspond to the steps you want to include in the main report file.
 - b. Paste the data you copied into the Steps section of the **Raw Data** worksheet of the main report file.
4. In the Raw Data worksheet of the main report file, click the **Regenerate Processed Data and Graphs** button update and overwrite the existing data and graphs in the Step, Sequence, and Sequence File worksheets.

Reports the Basic Step Time Report Generator Plug-In Creates

The Basic Step Time Report Generator plug-in generates a single report file for each UUT in , , and formats using the following filename format:

```
[Station ID][Sequence File Name][Test Socket Index][Batch
Serial Number][UUT Serial
Number][Time][Date].[html|csv|xls]
```

Microsoft Windows and Excel limit the number of characters in a file path. If the length of the file path for generated reports is less than 218 characters, the Basic Step Time Report Generator plug-in generates reports in HTML, .csv, and Excel format. If the length of the file path for generated reports is between 218 and 260 characters, the Basic Step Time Report Generator plug-in generates reports only in HTML and .csv formats. If the length of the file path for generated reports exceeds 260 characters, the Basic Step Time Report Generator plug-in does not generate any report and returns an

error before executing the sequence file.

The Basic Step Time Report plug-in uses an underscore (`_`) in the place of the `#` character and invalid filename characters in a UUT or batch serial number.

HTML

TestStand displays the HTML report in the Report pane, and the HTML report links to the `.CSV` and Excel reports.

The HTML report includes the following information for each UUT in the report header:

- Station ID
- Operator
- UUT Part Number (if non-empty)
- UUT Serial Number
- Batch Serial Number (if using the Batch model)
- TestSocket Index (if using the Batch or Parallel model)
- Date
- Time
- Execution Time
- UUT Result
- TSR Filename (if using the TestStand Offline Results Processing Utility)
- TSR ID (if using the TestStand Offline Results Processing Utility)
- TSR File Closed (if using the TestStand Offline Results Processing Utility)

.CSV

The `.CSV` report includes the following information for each UUT in the report header:

- Station ID
- Operator
- UUT Part Number (if non-empty)
- UUT Serial Number
- Batch Serial Number (if using the Batch model)
- TestSocket Index (if using the Batch or Parallel model)
- Date
- Time

- Execution Time
- UUT Result
- CSV File Format Version
- TSR Filename (if using the TestStand Offline Results Processing Utility)
- TSR ID (if using the TestStand Offline Results Processing Utility)
- TSR File Closed (if using the TestStand Offline Results Processing Utility)

For each step, the report includes the following information:

- Sequence Filename—Name of the sequence file that contains the step
- Sequence Name—Name of the sequence that contains the step
- Step Group—Name of the step group that contains the step
- Step ID—Unique ID of the step
- Step Name—Name of the step
- Parent ID—ID of the parent sequence call step
- Step Type—Type of the step
- ID—Unique execution ID of the step
- Status—Execution status of the step
- Start Time—Execution start time of the step
- Total Time—Total time the step takes to complete execution
- Module Time—Time the code module of the step takes or empty if the step uses the None adapter
- Sequence Call Sequence Filename—Path of the sequence file a non-asynchronous Sequence Call step calls
- Sequence Call Sequence Name—Name of the sequence a non-asynchronous Sequence Call step calls
- Number of Loops—Number of loops the step executes or empty if the step does not use looping
- Ending Loop Index—Index of the last iteration of the loop for the step or empty if the step does not use looping
- Loop Index—Index of the current iteration of the loop for the step or empty if the step does not use looping
- UUT Serial Number—Serial number of the UUT
- Batch Serial Number—Serial number of the batch that contains the UUT
- TestSocket Index—Index of the test socket the UUT uses



Note The Sequence File, Sequence, and Parent ID columns are empty for the

MainSequence Callback step from process model files.

Excel

The Excel report includes the following information for each UUT in an Excel workbook with the following worksheets:

- **Step**—Contains the following processed data for the steps used to test the UUT and graphs to show the average total time of the steps and the cumulative total time of the steps. You can modify the data and graphs in this worksheet.
 - Step Name
 - Step ID
 - Cumulative Total Time (of the step)
 - Cumulative Module Time (of the step)
 - Number of Execution
 - Average Total Time
 - Average Module Time
 - Max Total Time
 - Max Module Time
 - Min Total Time
 - Min Module Time



Note This worksheet does not contain information for the MainSequence Callback step from process model files.

- **Sequence**—Contains the following processed data for the sequences used to test the UUT and graphs to show the average total time of the sequences and the percentage of the cumulative total time of the sequences. You can modify the data and graphs in this worksheet.
 - Sequence File Name
 - Sequence Name
 - Cumulative Total Time (of each Sequence Call step that calls the sequence, including subsequence time)
 - Cumulative Total Time (of each Sequence Call step that calls the sequence, excluding subsequence time)
 - Number of Times Sequence Call Steps Call into the Sequence
 - Average Total Time (excluding subsequence time)

- Max Total Time (excluding subsequence time)
- Min Total Time (excluding subsequence time)



Note This worksheet contains information only for non-asynchronous Sequence Call steps. You can modify the source code to insert data for Post Action Sequence Call, callback sequence, and asynchronous Sequence Call steps.

- Sequence File—Contains the following processed data for the sequence files used to test the UUT and graphs to show the cumulative total time of the sequence files and the percentage of total time for the sequence files. You can modify the data and graphs in this worksheet.
 - Sequence Filename
 - Cumulative Total Time (of Sequence Call steps calling any sequence in the sequence file)
- Raw Data—Contains all the data from the `.csv` file in protected mode. Click the **Regenerate Processed Data and Graphs** button to update and overwrite the existing data and graphs in the Step, Sequence, and Sequence File worksheets.

Related concepts:

- [TestStand Directory Structure](#)
- [Process Model Plug-In Architecture](#)

Model Plug-in - Simple Text Report

<TestStand Public>\Examples\Customizing Result Processing\ModelPlugin Simple Text Report

Purpose

This example demonstrates how to implement a result processing process model plug-in using LabVIEW, LabWindows/CVI, and Microsoft Visual C#.

Example File Location

<TestStand Public>\Examples\Customizing Result Processing\
ModelPlugin Simple Text Report\Install Simple Text Report

Plugin.seq

Highlighted Features

Process model plug-in architecture

Major API

N/A

Prerequisites

None

How to Use This Example

You can use the <TestStand Public>\Examples\Customizing Result Processing\ModelPlugin Simple Text Report\SetupExamplePlugin.seq example sequence file to install a result processing process model plug-in that generates a simple text report you can use for system monitoring and debugging. NI recommends using the Microsoft Visual C# version if you intend to use the simple text reporting functionality only as provided.

LabVIEW

Complete the following steps to build and use the LabVIEW version of this example.

1. Open the <TestStand Public>\Examples\Customizing Result Processing\ModelPlugin Simple Text Report\LabVIEW\SimpleTextPlugin.lvproj file in LabVIEW.
2. Build the **Build Simple Text Plugin** build specification.
3. Confirm that the following files exist:
 - <TestStand Public>\Components\Models\ModelPlugins\NI_SimpleTextReport_LabVIEW\SimpleTextPlugin.llb
 - <TestStand Public>\Components\Models\ModelPlugins\NI_SimpleTextReport_LabVIEW.seq
4. Add and enable an instance of the LabVIEW simple text report process model plug-in in the Result Processing dialog box.

LabWindows/CVI

Complete the following steps to build and use the LabWindows/CVI version of this example.

1. Open the <TestStand Public>Examples\Customizing Result Processing\Model Plugin - Simple Text Report\CVI\SimpleTextReport.cws file in LabWindows/CVI.
2. Select **Build»Create Debuggable Dynamic Link Library**.
3. Copy the <TestStand Public>\Examples\Customizing Result Processing\ModelPlugin Simple Text Report\CVI\NI_SimpleTextReport_CVI.seq file to the <TestStand Public>\Components\Models\ModelPlugins directory.
4. Confirm that the <TestStand Public>\Components\Models\ModelPlugins\NI_SimpleTextReport_CVI\NI_SimpleTextReport_CVI.dll file exists and has been overwritten by the version you just built.
5. Add and enable an instance of the LabWindows/CVI simple text report process model plug-in in the Result Processing dialog box.

Microsoft Visual C#

Complete the following steps to build and use the Microsoft Visual C# version of this example.

1. Open and build the <TestStand Public>\Examples\Customizing Result Processing\ModelPlugin Simple Text Report\DotNet\SimpleTextReport.sln file.
2. Copy the <TestStand Public>\Examples\Customizing Result Processing\ModelPlugin Simple Text Report\DotNet\NI_SimpleTextReport_DotNet.seq file to the <TestStand Public>\Components\Models\ModelPlugins directory.
3. Create the <TestStand Public>\Components\Models\ModelPlugins\NI_SimpleTextReport_DotNet directory if it does not already exist.
4. Copy the <TestStand Public>\Examples\Customizing Result Processing\ModelPlugin Simple Text Report\DotNet\

SimpleTextPlugin_DotNet.dll file to the <TestStand Public>\Components\Models\ModelPlugins\NI_SimpleTextReport_DotNet directory.

5. Copy the <TestStand Public>\Examples\Customizing Result Processing\ModelPlugin Simple Text Report\DotNet\NationalInstruments.TestStand.Utility.dll file to the <TestStand Public>\Components\Models\ModelPlugins\NI_SimpleTextReport_DotNet directory.
6. Add and enable an instance of the Microsoft Visual C# simple text report process model plug-in in the Result Processing dialog box.

Related concepts:

- [TestStand Directory Structure](#)
- [Process Model Plug-In Architecture](#)

Examples for Demos

The Demo examples provide an overview of TestStand features. You can execute these examples without installing additional software. This directory includes the following example programs.

Computer Motherboard Test

The <TestStand Public>\Examples\Demos\Computer Motherboard Test directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Computer Motherboard Test Sequence - HTBasic

Purpose

This example simulates the testing of a computer motherboard using test steps that use the HTBasic Adapter. The example calls subsequences in the CPU Diagnostic

Sequence.seq and CPU Test Sequence.seq sequence files.

Example File Location

```
<TestStand Public>\Examples\Demos\Computer Motherboard  
Test\HTBasic\Computer Motherboard Test Sequence.seq
```

Highlighted Features

- Preconditions
- Sequence Call steps
- Test step types

Major API

None

Prerequisites

The example contains sequence files designed to work with HTBasic 8.0.

When you use the demo version of HTBasic, you must configure the HTBasic Adapter to use the HTBasic demo server. Select the **Use HTBasic Runtime Server** option in the HTBasic Adapter Configuration dialog box and specify the path and filename for HTBwin.exe.



Note (64-bit TestStand) 64-bit TestStand does not support the HTBasic Adapter.

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane of the `MainSequence`, review the steps in the Main step group.
 - The Powerup Test step, which is a Pass/Fail Test step, simulates a power-up test. Each of the five test steps that follow the Powerup Test specifies a

precondition expression that executes the step only when the Powerup Test step passes.

- The CPU Test step, which is a Sequence Call step, calls the `MainSequence` in the `CPU Test Sequence .seq` sequence file to simulate CPU tests.
- The ROM Test, RAM Test, Video Test, and Keyboard Test steps simulate tests. Each of the diagnostic steps
- The Powerup Diagnostics step, which is a Numeric Limit Test step, simulates a power-up diagnostics test. Each of the five diagnostic steps that follow the Powerup Diagnostics step specifies a precondition expression that executes the step only when the corresponding test step fails.
- The CPU Diagnostics step, which is a Sequence Call step, calls the `MainSequence` in the `CPU Diagnostic Sequence .seq` sequence file to simulate CPU diagnostics.

2. Select **Execute»Single Pass** to run the sequence.

3. When execution completes, review the report on the Report pane to see how TestStand reports the passed and failed steps.

Related concepts:

- [TestStand Directory Structure](#)
- [Step Groups](#)

Computer Motherboard Test Sequence - LabWindows/CVI

Purpose

This example simulates the testing of a computer motherboard using test steps that use the LabWindows/CVI Adapter.

Example File Location

```
<TestStand Public>\Examples\Demos\Computer Motherboard
Test\CVI\Computer Motherboard Test Sequence.seq
```

Highlighted Features

- Flow Control step types
- Preconditions

- Sequence Call steps
- Test step types

Major API

None

Prerequisites

You must have the LabWindows/CVI Run-Time Engine installed and you must configure the LabWindows/CVI Adapter to execute steps in-process. If you want to use the **Execute Steps in an External Instance of CVI** option, you must have the LabWindows/CVI development environment installed.



Note In order to debug the example code module, you must first rebuild the source files in the debug configuration within LabWindows/CVI.

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane of the `MainSequence`, review the steps in the Setup step group.
 - The Simulation Dialog step calls a DLL that launches a dialog box to prompt you to select a test or tests you want to simulate to fail.
 - The Turn Vacuum Table On step, which is an Action step, simulates the activation of a vacuum table. In the Cleanup step group, the Turn Vacuum Table Off step deactivates the vacuum table.
2. On the Steps pane, review the steps in the Main step group.
 - The Powerup Test step, which is a Pass/Fail Test step, simulates a power-up test, and the results of the step determine whether the following If/Else structure continues testing or executes the Powerup Diagnostics step.
 - The CPU Test step, which is a Sequence Call step, calls the CPU Test subsequence to simulate CPU tests.
 - The ROM Test, RAM Test, Video Test, and Keyboard Test steps simulate tests, and if any test fails, the following If structure executes.
 - Each step in the If structure specifies a precondition expression that executes

the step only when the corresponding test step fails.

3. Select **Execute»Single Pass** to run the sequence.
4. In the Test Simulator dialog box, select the **Power On** test to fail.
5. When execution completes, review the report on the Report pane. The report indicates that the test steps after the Powerup Test step did not execute, but the Powerup Diagnostics step did execute.
6. Select **Execute»Restart** to run the sequence again.
7. In the Test Simulator dialog box, select any test other than the Power On test to fail.
8. When execution completes, review the report. The report indicates that the Powerup Test step passes and TestStand ran the remaining tests, and that TestStand ran diagnostic steps for any failed test steps but skipped diagnostic steps for any passed test steps.

Related concepts:

- [Programming with the TestStand API in LabWindows/CVI](#)
- [TestStand Directory Structure](#)
- [Step Groups](#)

Computer Motherboard Test Sequence - LabVIEW

Purpose

This example simulates the testing of a computer motherboard using test steps that use the LabVIEW Adapter.

Example File Location

```
<TestStand Public>\Examples\Demos\Computer Motherboard
Test\LabVIEW\Computer Motherboard Test Sequence.seq
```

Highlighted Features

- Flow Control step types
- Preconditions
- Sequence Call steps
- Test step types

Major API

None

Prerequisites

None



Note If you configure the LabVIEW adapter to use the development system and then run the example, you may receive an error indicating that the VIs are saved in a later version of LabVIEW. In this case, you can rebuild the example VIs using the source project located in the source subdirectory of the example directory. To rebuild the example files, open the project and rebuild the source distribution in the build specifications section.

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane of the `MainSequence`, review the steps in the Setup step group.
 - The Simulation Dialog step calls a VI that launches a dialog box to prompt you to select a test or tests you want to simulate to fail.
 - The Turn Vacuum Table On step, which is an Action step, simulates the activation of a vacuum table. In the Cleanup step group, the Turn Vacuum Table Off step deactivates the vacuum table.
2. On the Steps pane, review the steps in the Main step group.
 - The Powerup Test step, which is a Pass/Fail Test step, simulates a power-up test, and the results of the step determine whether the following If/Else structure continues testing or executes the Powerup Diagnostics step.
 - The CPU Test step, which is a Sequence Call step, calls the CPU Test subsequence to simulate CPU tests.
 - The ROM Test, RAM Test, Video Test, and Keyboard Test steps simulate tests, and if any test fails, the following If structure executes.
 - Each step in the If structure specifies a precondition expression that executes the step only when the corresponding test step fails.
3. Select **Execute»Single Pass** to run the sequence.

4. In the Test Simulator dialog box, select the **Power On** test to fail.
5. When execution completes, review the report on the Report pane. The report indicates that the test steps after the Powerup Test step did not execute, but the Powerup Diagnostics step did execute.
6. Select **Execute»Restart** to run the sequence again.
7. In the Test Simulator dialog box, select any test other than the Power On test to fail.
8. When execution completes, review the report. The report indicates that the Powerup Test step passes and TestStand ran the remaining tests, and that TestStand ran diagnostic steps for any failed test steps but skipped diagnostic steps for any passed test steps.

Related concepts:

- [Programming with the TestStand API in LabVIEW](#)
- [TestStand Directory Structure](#)
- [Step Groups](#)

Computer Motherboard Test Sequence - .NET

Purpose

This example simulates the testing of a computer motherboard using test steps that use the .NET Adapter.

Example File Location

```
<TestStand Public>\Examples\Demos\Computer Motherboard  
Test\DotNET\Computer Motherboard Test Sequence.seq
```

Highlighted Features

- Flow Control step types
- Preconditions
- Sequence Call steps
- Test step types

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane of the `MainSequence`, review the steps in the Setup step group.
 - The Simulation Dialog step calls a .NET assembly that launches a dialog box to prompt you to select a test or tests you want to simulate to fail.
 - The Turn Vacuum Table On step, which is an Action step, simulates the activation of a vacuum table. In the Cleanup step group, the Turn Vacuum Table Off step deactivates the vacuum table.
2. On the Steps pane, review the steps in the Main step group.
 - The Powerup Test step, which is a Pass/Fail Test step, simulates a power-up test, and the results of the step determine whether the following If/Else structure continues testing or executes the Powerup Diagnostics step.
 - The CPU Test step, which is a Sequence Call step, calls the CPU Test subsequence to simulate CPU tests.
 - The ROM Test, RAM Test, Video Test, and Keyboard Test steps simulate tests, and if any test fails, the following If structure executes.
 - Each step in the If structure specifies a precondition expression that executes the step only when the corresponding test step fails.
3. Select **Execute»Single Pass** to run the sequence.
4. In the Test Simulator dialog box, select the **Power On** test to fail.
5. When execution completes, review the report on the Report pane. The report indicates that the test steps after the Powerup Test step did not execute, but the Powerup Diagnostics step did execute.
6. Select **Execute»Restart** to run the sequence again.
7. In the Test Simulator dialog box, select any test other than the Power On test to fail.
8. When execution completes, review the report. The report indicates that the

Powerup Test step passes and TestStand ran the remaining tests, and that TestStand ran diagnostic steps for any failed test steps but skipped diagnostic steps for any passed test steps.

Related concepts:

- [TestStand Directory Structure](#)
- [Step Groups](#)

Computer Motherboard Test Sequence - Python

Purpose

This example simulates the testing of a computer motherboard using test steps that use the Python Adapter.

Example File Location

```
<TestStand Public>\Examples\Demos\Computer Motherboard  
Test\Python\Computer Motherboard Test Sequence.seq
```

Highlighted Features

- Flow Control step types
- Preconditions
- Post Actions
- Sequence Call steps
- Test step types

Major API

None

Prerequisites

You must have the required version of CPython interpreter installed and added to your PATH variable. You must update the Python version to use in the Python Adapter Configuration dialog box before using the example.

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane of the MainSequence sequence, review the steps in the Setup step group.
 - The Simulation Dialog step calls a CVI DLL that launches a dialog box to prompt you to select a test or tests you want to simulate to fail.
 - The Initialize Computer Test step, which is an Action step, creates an instance of a Python class that stores the settings the user specifies in the simulation dialog. TestStand uses the instance of the class to call class functions which do the testing.
 - The Turn Vacuum Table On step, which is an Action step, simulates the activation of a vacuum table. In the Cleanup step group, the Turn Vacuum Table Off step deactivates the vacuum table.
2. On the Steps pane, review the steps in the Main step group.
 - The Powerup Test step, which is a Pass/Fail Test step, simulates a power-up test, and the results of the step determine whether the following If/Else structure continues testing or executes the Powerup Diagnostics step.
 - The CPU Test step, which is a Sequence Call step, calls the CPU Test subsequence to simulate CPU tests.
 - The ROM Test, RAM Test, Video Test, and Keyboard Test steps simulate tests, and if any test fails, the following If structure executes.
 - Each step in the If structure specifies a precondition expression that executes the step only when the corresponding test step fails.
3. Select **Execute»Single Pass** to run the sequence.
4. In the Test Simulator dialog box, select the **Power On** test to fail.
5. When execution completes, review the report on the Report pane. The report indicates that the test steps after the Powerup Test step did not execute, but the Powerup Diagnostics step did execute.
6. Select **Execute»Restart** to run the sequence again.
7. In the Test Simulator dialog box, select any test other than the Power On test to fail.
8. When execution completes, review the report. The report indicates that the Powerup Test step passes and TestStand ran the remaining tests, and that TestStand ran diagnostic steps for any failed test steps but skipped diagnostic steps for any passed test steps.

Related concepts:

- [TestStand Directory Structure](#)
- [Step Groups](#)

Mobile Device Test Demo

Purpose

This example simulates the testing of a cellular phone with test steps that use the LabVIEW, LabWindows/CVI, and .NET Adapters.

Example File Location

```
<TestStand Public>\Examples\Demos\Mobile Device Test\Mobile  
Device Test.seq
```

Highlighted Features

- Test step types
- Flow Control step types
- Report generation

Major API

None

Prerequisites

None



Note If you configure the LabVIEW adapter to use the development system and then run the example, you may receive an error indicating that the VIs are saved in a later version of LabVIEW. In this case, you can rebuild the example VIs using the source project located in the source subdirectory of the example directory. To rebuild the example files, open the project and rebuild the source distribution in the build specifications section.

How to Use This Example

Complete the following steps to review the sequences and steps in the example.

1. Select the **MainSequence** in the Sequences pane and review the following steps in the Steps pane:
 - The Simulation Dialog step launches a dialog box to prompt you to select a test or tests you want to simulate to fail.
 - Notice that the test steps are configured to use different adapters based on the implementation of the code module.
2. Many of the tests are contained within a subsequence and are called using sequence call steps. Double-click a sequence call step to view the subsequence that it calls.
3. The code modules in this example implement simulated hardware calls and test functionality. Navigate to the `<TestStand Public>\Examples\Demos\Mobile Device Test` directory to view the source code for the code modules.

Complete the following steps to run the example.

1. Select **Execute»Single Pass** to run the sequence.
2. Select a UUT configuration in the dialog box.
3. When execution completes, review the report on the Report pane. The report for this example has been customized to demonstrate the following TestStand features for report customization:
 - The audio and user input test steps use the additional results feature to log the simulated data to the report.
 - The LCD Video and RF test steps use the ReportText field to log custom HTML code that contains an image generated in the code module.

You can run this example using the Batch or Parallel process models. Complete the following steps to run the example with a different process model.

1. Select **Edit»Sequence File Properties**.
2. On the Advanced tab, select **Require Specific Model** in the Model Option field.
3. Select **ParallelModel.seq** or **BatchModel.seq** in the Model File control.
4. Click **OK** to close the dialog box.
5. Select **Execute»Single Pass** to run the sequence. Notice that multiple instances of the test now execute.

Related concepts:

- [Programming with the TestStand API in LabVIEW](#)
- [Programming with the TestStand API in LabWindows/CVI](#)
- [TestStand Directory Structure](#)

Parallel Test Strategies Demo

Purpose

This example demonstrates how you can use built-in parallel testing features to improve test throughput. You can use the parallel or batch process models to test multiple units in parallel on a single test station. TestStand provides a set of synchronization steps to coordinate parallel and batch testing.

You can use the Auto Schedule step type to further improve the performance of parallel tests by allowing TestStand to automatically reorder your test steps to better utilize hardware resources.

This example shows the effects of using these features and how they can improve the performance of your tests.

Example File Location

```
<TestStand Public>\Examples\Demos\Parallel Testing  
Strategies\Parallel Testing Strategies.seq
```

Highlighted Features

- Batch process model
- Auto Schedule step type

Major API

None

Prerequisites

None

How to Use This Example

To begin the demo, select **Execute»Run MainSequence**.

When the example starts, a new window appears to show the simulated test steps, the current instrument usage, and the total test time for each strategy so you can compare performance. The demo contains the following strategies:

- **Sequential**—In turn, each UUT performs a DMM test. Then, in turn, each UUT performs a Scope test.
- **Parallel**—In parallel, each UUT performs a DMM test and then a Scope test. A UUT locks each instrument it uses to ensure that it has exclusive use of the instrument.
- **Auto-Scheduled**—In parallel, each UUT performs a DMM test and then a Scope test. A UUT locks each instrument it uses to ensure that it has exclusive use of the instrument. The order in which the UUT performs the test varies depending on the availability of the instruments.
- **Parallel with Additional DMM**—In parallel, each UUT performs a DMM test using the first available DMM. Then, each UUT performs a Scope test. A UUT locks each instrument it uses to ensure that it has exclusive use of the instrument.
- **Auto-Scheduled with Additional DMM**—In parallel, each UUT performs a DMM test using the first available DMM. Then, each UUT performs a Scope test. A UUT locks each instrument it uses to ensure that it has exclusive use of the instrument. The order in which the UUT performs the tests varies depending on the availability of the instruments.



Note The first time you execute this example, the auto-scheduled tests might take longer to execute than the parallel tests due to components specific to the example. In this case, running the example a second time provides a more accurate representation of the performance differences.

Related concepts:

- [Parallel Process Model](#)
- [Batch Process Model](#)
- [TestStand Directory Structure](#)

Examples for Fundamentals

The Fundamentals examples demonstrate key concepts for developing TestStand systems. This directory includes the following example programs.

Calling Dynamic Dispatch VIs

Purpose

This example demonstrates how you can use dynamic dispatch with LabVIEW classes directly from a TestStand sequence to implement a hardware abstraction layer (HAL). In this approach, TestStand calls the implementation of each Class VI corresponding to the class you select in the initial dialog.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Calling Dynamic  
Dispatch VIs\Calling Dynamic Dispatch VIs.seq
```

Highlighted Features

- LabVIEW Adapter - Class Member Call
- Dynamic Dispatch

Major API

None

Prerequisites

The LabVIEW 2012 Run-Time engine or LabVIEW 2012 or higher development system is required to execute this example. Dynamic dispatch support in TestStand requires LabVIEW 2012 or higher.

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to execute the sequence.
2. When prompted, select the DMM base class and click **OK**.
3. When the sequence completes, view the report. Notice that the report text indicates the class used.
4. Select **Execute»Single Pass** to execute the sequence again.
5. When prompted, select one of the child classes and click **OK**.
6. When the sequence completes, view the report. Notice that the report text has now changed because the steps called the specific VIs in the class you selected.

Complete the following steps to review the sequences and steps in this example:

1. In MainSequence, select the **Select Class VI**. This VI returns a reference to a different LabVIEW class based on the user selection. The VI can return the parent DMM Base class, or one of two child classes—PCI-4070 or Simulated DMM.



Note The Initialize, Read, and Close steps are configured using the Class member call call type in order to call members of the Generic DMM class. Based on the class selected previously, TestStand dynamically selects the implementation of the class VI which matches the class reference. This action allows the sequence to behave differently based on the type of LabVIEW class reference selected.

2. To view the implementation of the LabVIEW classes, open classes.prj, located in `<TestStand Public>\Examples\Fundamentals\Calling Dynamic Dispatch VIs`.

Related concepts:

- [TestStand Directory Structure](#)
- [Calling LabVIEW Class Member VIs from TestStand](#)
- [Passing an Instance of a LabVIEW Class Object](#)

Developing Platform Independent Test Systems

Purpose

This example demonstrates how you can create test sequences that can execute in 32-bit TestStand and 64-bit TestStand with no modifications.

The example uses the `$(Platform)` path macro in the module path to call the correct version of a LabWindows/CVI DLL and a C++ DLL based on the current bitness of TestStand.

The example uses a LabVIEW code module with the Separate Compiled Code from source file option enabled so 32-bit LabVIEW and 64-bit LabVIEW can execute the VI.

The example assembly is built using the Any CPU configuration so that 32-bit TestStand and 64-bit TestStand can call it.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Developing  
Platform Independent Test Systems\Developing Platform  
Independent Test Systems.seq
```

Highlighted Features

- `$(Platform)` path macro
- Separate Compiled Code from Source File LabVIEW option
- Any CPU Visual Studio Build Configuration setting

Major API

Engine.Is64Bit property

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select the **CVI** step or the **C++ Action** step. Observe that the `$(Platform)` path macro is present in the Module path. This macro evaluates as `win32` or `x64` in the resolved path based on the current TestStand bitness.
2. Select the **.NET Action** step and click **Edit** to open the VI. Notice that the same code module path will be used for 32-bit TestStand and 64-bit TestStand. Because the

assembly is built using the Any CPU Visual Studio Build Configuration Setting, the module can be called by either bitness of TestStand.

3. If the LabVIEW development system is installed, select the **LabVIEW Action** step and click **Edit** to open the VI. In LabVIEW, select **File»VI Properties**. Notice that the **Separate Compiled code from source file** option is enabled. This setting allows the same VI to execute in 32-bit LabVIEW and 64-bit LabVIEW.
4. Observe that the IF statement uses the `Engine.Is64Bit` property to determine the current bitness of TestStand. This property allows you to define different behavior for 32-bit TestStand and 64-bit the bitness of TestStand.
5. Run the example and notice that the executed code module bitness matches TestStand.
6. If possible, open the example in the other bitness of TestStand and observe the difference in behavior.

Related concepts:

- [TestStand Directory Structure](#)
- [Using the \\$\(Platform\) Path Macro to Locate the Correct Code Module in 32-bit TestStand and 64-bit TestStand](#)
- [Executing VIs with Separate Compiled Code](#)
- [AnyCPU Assemblies in 32-bit TestStand and 64-bit TestStand](#)

Launching a Modal Dialog

The <TestStand Public>\Examples\Fundamentals\Launching a Modal Dialog directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Launching a Modal Dialog - LabVIEW

Purpose

This example demonstrates how to use LabVIEW to create a dialog box that is modal to the TestStand main application window.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Launching a Modal  
Dialog\LabVIEW\Launching a Modal Dialog.seq
```

Highlighted Features

- LabVIEW Adapter
- Modal dialog boxes

Major API

None

Prerequisites

To run this example, you must have the LabVIEW development system installed.



Note The Modal Dialog palette VIs do not require the LabVIEW development system. This requirement applies only to this example.

How to Use This Example

This example uses a LabVIEW Action step to call VIs configured to function as modal dialog boxes, just as if the step were calling any other VI.

To review how the VI must be configured, open the `dialog.vi` located in the `<TestStand Public>\Examples\Fundamentals\Launching a Modal Dialog\LabVIEW` directory. In the VI, the modal dialog starts with the Start Modal Dialog VI and ends with the End Modal Dialog VI. Each VI requires the sequence context from the calling TestStand sequence.

Review the instructions documented on the block diagram of the example VI to achieve proper functionality.

Related concepts:

- [TestStand Directory Structure](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Making Dialog Boxes Modal to TestStand](#)
- [Sequence Context](#)

Launching a Modal Dialog - MFC

Purpose

This example demonstrates how to use C++ and the TestStand API to create a dialog box that is modal to the TestStand main application window.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Launching a Modal  
Dialog\MFC\Launching a Modal Dialog.seq
```

Highlighted Features

- Modal dialog boxes
- TestStand API

Major API

- Engine.NotifyStartOfModalDialogEx
- Engine.NotifyEndOfModalDialog

Prerequisites

C++ compiler

How to Use This Example

This example uses a C/C++ DLL Action step to call a function in a C++ DLL to create a modal dialog box.

To review how the function behaves, open `ExampleModalDlg.cpp` located in the `<TestStand Public>\Examples\Fundamentals\Launching a Modal`

Dialog\MFC directory. The function calls the `Engine.NotifyStartOfModalDialogEx` method, which instructs the TestStand Engine that the DLL is creating a modal dialog box. After the dialog box is dismissed, the function calls the `Engine.NotifyEndOfModalDialog` method to end the modal dialog box state.

Related concepts:

- [TestStand Directory Structure](#)
- [Making Dialog Boxes Modal to TestStand](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)

Launching a Modal Dialog - .NET

Purpose

This example demonstrates how to use .NET and the TestStand API to create a dialog box that is modal to the TestStand main application window.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Launching a Modal  
Dialog\DotNet\Launching a Modal Dialog.seq
```

Highlighted Features

- Modal dialog boxes
- TestStand API

Major API

- `Engine.NotifyStartOfModalDialogEx`
- `Engine.NotifyEndOfModalDialog`

Prerequisites

None

How to Use This Example

This example uses a .NET Action step to call a function in a .NET DLL to create a modal dialog box.

Related concepts:

- [TestStand Directory Structure](#)
- [Making Dialog Boxes Modal to TestStand](#)

Launching an MFC Dialog with ActiveX Controls

Purpose

This example demonstrates how to launch a Microsoft Foundation Classes (MFC) dialog box that contains an ActiveX control when you call the MFC DLL from TestStand.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Launching an MFC  
dialog With ActiveX Controls\Launching an MFC dialog With  
ActiveX Controls.seq
```

Highlighted Features

Sequence Call step

Major API

N/A

Prerequisites

None

How to Use This Example

This example shows how to create and properly initialize a thread so it can launch an MFC dialog box that contains ActiveX controls. TestStand must initialize all execution

threads for the ActiveX multithreaded concurrency model. MFC requires that any thread that launches a dialog box that contains ActiveX controls must be initialized as apartment-threaded. Therefore, to launch such a dialog box, you must create a temporary worker thread you initialize as apartment-threaded and have the worker launch the dialog box. Meanwhile, the original thread TestStand uses to call the function must wait for the worker thread to exit.



Note When an MFC DLL launches a dialog box that contains an ActiveX control and the DLL does not create a worker thread initialized as apartment-threaded, you can call the DLL function from a TestStand execution thread initialized as apartment-threaded. Use a Sequence Call step to create a new thread and enable the **Use Thread Apartment** setting in the Sequence Call Advanced Settings dialog box.

Related concepts:

- [TestStand Directory Structure](#)

Overriding Engine Callbacks - SequenceFilePostStepFailure

Purpose

This example demonstrates how to retry or force failed steps to pass using the SequenceFilePostStepFailure Engine callback sequence. You can force steps in the MainSequence or in a subsequence.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Overriding Engine  
Callbacks\Overriding SequenceFilePostStepFailure  
Callback.seq
```

Highlighted Features

Callbacks

Major API

Step.CancelStepCallback

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequences and steps in this example.

1. On the Sequences pane, select the `MainSequence`. The `MainSequence` contains two Numeric Limit Test steps and one Sequence Call step.
2. On the Sequences pane, select the **SubSequence** sequence. The `SubSequence` sequence contains Numeric Limit Test steps, three of which are designed to fail in the following ways to demonstrate how TestStand behaves when a step fails:
 - One test fails only the first time
 - One test fails every time
 - One test fails every time but does not cause the sequence to fail
3. On the Sequences pane, select the **SequenceFilePostStepFailure** callback. The first step, which uses the ActiveX/COM Adapter, sets the `Step.CancelStepCallback` property to `True` to suppress any other failure callbacks for a failed step, and a series of Statement steps collects information about the step failure.
4. On the Steps pane, select the **Choose Action** step.
5. On the Step Settings pane, click the Text and Buttons tab. The Message Expression control specifies an expression that provides information about the step failure, and the step specifies buttons so you can choose to retry the test, force the test to pass, continue the execution, or terminate the execution. The Select step that follows the Choose Action step specifies the following cases that correspond to the buttons you can click in the message popup that the Choose Action step creates:
 - Case 1 changes the next step index to retry the step that failed
 - Case 2 forces the failed step to pass
 - Case 3 continues the sequence without taking any other action
 - Case 4 terminates the execution

Complete the following steps to run the example.

1. Select **Execute»Single Pass** to run the sequence.
2. TestStand launches the Step Failed dialog box for the first step that fails. Click the **Retry** button to run the failed step again. The step passes on the second attempt.
3. TestStand launches the Step Failed dialog box for the next step that fails. Click the **Force to Pass** button to force the failed step to pass.
4. TestStand launches the Step Failed dialog box for the next step that fails. Click the **Continue** button to allow the step to fail without causing the sequence fail.
5. When execution completes, review the report on the Report pane. The status of the first attempt of the Fail only the first time step is Failed, Retrying Step because you clicked the **Retry** button after the step failed once. The status of the Fail every time step is Passed because you clicked the **Force to Pass** button when the step failed.

Related concepts:

- [TestStand Directory Structure](#)
- [Callback Sequences](#)

Overriding Engine Callbacks - SequenceFilePostStepRuntimeError

Purpose

This example demonstrates how to handle run-time errors using the SequenceFilePostStepRuntimeError Engine callback sequence.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Overriding Engine  
Callbacks\Overriding SequenceFilePostStepRuntimeError  
Callback.seq
```

Highlighted Features

Callbacks

Major API

Step.CancelStepCallback

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequences and steps in this example.

1. On the Sequences pane, select the `MainSequence`. The `MainSequence` contains two Numeric Limit Test steps and one Sequence Call step.
2. On the Sequences pane, select the **SubSequence** sequence. The `SubSequence` sequence contains Numeric Limit Test steps, two of which are designed to cause run-time errors in the following ways to demonstrate how TestStand behaves when a run-time error occurs:
 - One test generates an error only the first time
 - One test generates an error every time
3. On the Sequences pane, select the **SequenceFilePostStepRuntimeError** callback. The first step, which uses the ActiveX/COM Adapter, sets the `Step.CancelStepCallback` property to `True` to suppress any other failure callbacks for a failed step.
4. On the Steps pane, select the **Choose Action** step.
5. On the Step Settings pane, click the Text and Buttons tab. The Message Expression control specifies an expression that provides information about the step that caused the run-time error, and the step specifies buttons so you can choose to retry the test, ignore the error, continue to the Cleanup step group, or terminate the execution. The Select step that follows the Choose Action step specifies the following cases that correspond to the buttons you can click in the message popup that the Choose Action step creates:
 - Case 1 ignores the error and changes the next step index to retry the step that caused the error
 - Case 2 ignores the error and specifies the step status as `Error Suppressed`.
 - Case 3 sets the `RunState.Caller.Runstate.ErrorReported` flag to `True` to prevent TestStand from displaying a run-time error dialog box because has already reported the error
 - Case 4 sets the `RunState.Caller.Runstate.ErrorReported` flag to `True` to prevent TestStand from displaying a run-time error dialog box because has already reported the error and then terminates the execution

Complete the following steps to run the example.

1. Select **Execute»Single Pass** to run the sequence.
2. TestStand launches the Error dialog box for the first step that causes a run-time error. Click the **Retry** button to run the step again. The step passes on the second attempt.
3. TestStand launches the Error dialog box for the next step that causes a run-time error. Click the **Ignore Error** button to continue executing the sequence.
4. When execution completes, review the report on the Report pane. The status of the first attempt of the Generate a run-time error only the first time step is Error, Retrying Step because you clicked the **Retry** button after the step caused a run-time error once. The status of the Generate a run-time error every time step is Error Suppressed because you clicked the **Ignore Error** button when the step caused a run-time error.

Related concepts:

- [TestStand Directory Structure](#)
- [Callback Sequences](#)
- [Step Groups](#)

Passing Clusters to Code Modules - LabVIEW

This example demonstrates how to pass a TestStand container as a LabVIEW cluster to a LabVIEW VI. The container consists of several different data types. The VIs are compiled into a DLL and called as functions.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Passing Clusters  
to Code Modules\Passing Clusters to Code Modules.seq
```

Highlighted Features

- Data types
- LabVIEW Adapter
- Local variables

Major API

None

Prerequisites

You do not need to have the LabVIEW development system installed to use this example, but you must have the LabVIEW development system installed if you want to review the source VIs.

How to Use This Example

The sequence specifies `ContainerOut` and `ContainerIn` container local variables, which you can review on the Variables pane. When the sequence runs, TestStand copies the values in the `ContainerOut` container to the individual local variables, and then copies the values again to the `ContainerIn` container. A VI handles both operations. Before you run the sequence, the `ContainerOut` container contains values, the `ContainerIn` container contains default values, and the individual local variables contain default values.

Complete the following steps to use this example.

1. Select **Execute»Test UUTs** to run the sequence.
2. Enter any serial number in the UUT Information dialog box and click **OK**. TestStand launches a Results for Unbundle dialog box that lists values loaded from the individual local variables, which specified default values before you executed the sequence.
3. Click **OK** in the Results for Unbundle dialog box. TestStand launches another Results for Unbundle dialog box that lists values loaded from the `ContainerIn` container, which specified default values before you executed the sequence.
4. Click **OK** when TestStand informs you that the sequence passed.
5. Click **Stop** in the UUT Information dialog box to end the execution.

You can review the two Message Popup steps in the sequence to verify that the appropriate sources are providing values. You can open `VI1.vi` and `VI2.vi`, located in the `<TestStand Public>\Examples\Fundamentals\Passing Clusters to Code Modules` directory, to see how the data is being copied. The `VI1.vi` uses an `unbundle` function to separate the values from the cluster, and the

VI2.vi uses a bundle function to combine individual values into a single cluster.

To pass a TestStand container as a cluster to a VI, you must complete the following tasks:

- Define the container as a custom data type.
- Specify that objects of the data type can be passed as clusters.
- Ensure that the Cluster Item Label for each item in the data type matches the label of the LabVIEW cluster.

Complete the following steps to use the Types window to review the data type definition for a container.

1. In the Sequence File window, right-click the `ContainerIn` container on the Variables pane and select **View»Type Definition** from the context menu to launch the Types window.
2. Right-click the `ClusterType` data type and select **Properties** from the context menu to launch the Type Properties dialog box. Because the sequence is using a VI, you must enable cluster passing for the data type.
3. Click the Cluster Passing tab. Notice that the Allow Objects of This Type to be Passed as LabVIEW Clusters option is enabled. The Cluster Item Label option for each item matches the control labels in the VI.
4. Click **OK** to close the Type Properties dialog box.

Related concepts:

- [TestStand Directory Structure](#)
- [Data Types](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Sequence Local Variables](#)

Passing Structs to Code Modules

The `<TestStand Public>\Examples\Fundamentals\Passing Structs to Code Modules` directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Passing Structs to Code Modules - C

Purpose

This example demonstrates how to pass a TestStand container as a C-style struct to a function in a DLL. This example uses functions to pass the struct by both reference and value. For a DLL function to change the value of a TestStand container, you must pass the reference of the container to the function.



Note When you configure the container as a custom data type, be aware of the byte-packing the DLL function uses.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Passing Structs to  
Code Modules\C\Passing Structs to Code Modules.seq
```

Highlighted Features

- Calling DLLs from TestStand
- C/C++ DLL Adapter
- Local variables
- Data types

Major API

None

Prerequisites

C compiler or DLL

How to Use This Example

The sequence specifies `sPacked8Out` and `sPacked8In` container local variables,

which you can review on the Variables pane. The sequence uses a C DLL to copy the values from the `sPacked8Out` container to the `sPacked8In` container first by reference and then by value.

Complete the following steps to use this example.

1. Select **Execute»Test UUTs** to run the sequence.
2. Enter any serial number in the UUT Information dialog box and click **OK**. TestStand launches a Get Results dialog box that lists values loaded from the `sPacked8In` container, which specified default values before you executed the sequence.
3. Click **OK** in the Get Results dialog box. TestStand launches another Get Results dialog box that lists values loaded from the `sPacked8In` container. The `GetStructByRef2` step specifies a pre-expression that specifies a value of 0 for each variable, and then the `GetStructByRef2` step specifies new values for each variable.
4. Click **OK** when TestStand informs you that the sequence passed.
5. Click **Stop** in the UUT Information dialog box to end the execution.

You can review the two Message Popup steps in the sequence to verify that the `sPacked8In` container is displaying the variable values. The functions the DLL is using are available in `StructTest.cpp`, located in the `<TestStand Public>\Examples\Fundamentals\Passing Structs to Code Modules\C` directory.

To pass a TestStand container as a struct to a DLL, you must define the container as a custom data type, and you must specify that objects of the data type can be passed as structs.

Complete the following steps to use the Types window to review the data type definition for a container.

1. In the Sequence File window, right-click the `sPacked8In` container on the Variables pane and select **View»Type Definition** from the context menu to launch the Types window.
2. Right-click the `AllDataTypesPacked8` data type and select **Properties** from the context menu to launch the Type Properties dialog box. Because the sequence is using a DLL, you must enable C struct passing for the data type.
3. Click the C Struct Passing tab. Notice that the Allow Objects of This Type to be Passed as Structs option is enabled. The Packing ring control specifies the 8 Byte

Boundary option, which is the packing scheme the DLL uses.

4. Click **OK** to close the Type Properties dialog box.
5. In the Types window, expand the `AllDataTypesPacked8` data type. You can right-click each property and select **Properties** from the context menu to launch the Type Properties dialog box for each property of the data type. You can review each property to see how it is configured.

Related concepts:

- [TestStand Directory Structure](#)
- [Calling DLLs from TestStand](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)
- [Sequence Local Variables](#)
- [Data Types](#)

Passing Structs to Code Modules - LabWindows/CVI

Purpose

This example demonstrates how to pass a TestStand container as a C-style struct to a function in a LabWindows/CVI DLL. The container consists of several different data types. For a DLL function to change the value of a TestStand container, you must pass the reference of the container to the function.



Note When you configure the container as a custom data type, be aware of the byte-packing the DLL function uses. The default packing scheme in LabWindows/CVI can be either 1- or 8-byte, depending on the compatibility mode.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Passing Structs to  
Code Modules\CVI\Passing Structs to Code Modules.seq
```

Highlighted Features

- LabWindows/CVI Adapter

- Local variables
- Data types

Major API

None

Prerequisites

You must have the LabWindows/CVI Run-Time Engine installed and you must configure the LabWindows/CVI Adapter to execute steps in-process. If you want to use the **Execute Steps in an External Instance of CVI** option, you must have the LabWindows/CVI development environment installed.

How to Use This Example

The sequence specifies a `CVIStruct` container local variable, which you can review on the Variables pane. When the sequence runs, TestStand copies the values in the `CVIStruct` container to the individual local variables, and then assigns new values to the `CVIStruct` container. The LabWindows/CVI DLL handles both operations. Before you run the sequence, the `CVIStruct` container contains values, and the individual local variables contains default values.

Complete the following steps to use this example.

1. Select **Execute»Test UUTs** to run the sequence.
2. Enter any serial number in the UUT Information dialog box and click **OK**. TestStand launches a Results for Unpacking dialog box that lists values loaded from the `CVIStruct` container and assigned to the individual local variables.
3. Click **OK** in the Results for Unpacking dialog box. TestStand launches another Results dialog box that lists values modified and saved back to the `CVIStruct` container.
4. Click **OK** when TestStand informs you that the sequence passed.
5. Click **Stop** in the UUT Information dialog box to end the execution.

You can review the two Message Popup steps in the sequence to verify that the appropriate sources are providing values. You can open `CVIStructTestSource.c`, located in the `<TestStand Public>\Examples\Fundamentals\Passing`

Structs to Code Modules\CVI directory, to see how the data is being copied. The `PassStructAndUnpack` method accesses the values in the `CVIStruct` container and assigns the values to the individual method parameters. The `PassStructAndModify` method assigns arbitrary values to the elements of the `CVIStruct` container by reference. These assignments modify the container in TestStand. The sequence is using `CVIStructTest.dll`, into which the `PassStructAndUnpack` and `PassStructAndModify` methods were built.

To pass a TestStand container as a struct to a DLL, you must define the container as a custom data type, and you must specify that objects of the data type can be passed as structs.

Complete the following steps to use the Types window to review the data type definition for a container.

1. In the Sequence File window, right-click the `CVIStruct` container on the Variables pane and select **View»Type Definition** from the context menu to launch the Types window.
2. Right-click the `CVIStructExample` data type and select **Properties** from the context menu to launch the Type Properties dialog box. Because the sequence is using a DLL, you must enable C struct passing for the data type.
3. Click the C Struct Passing tab. Notice that the Allow Objects of This Type to be Passed as Structs option is enabled. The Packing ring control specifies the Default Adapter Packing option, which is the packing scheme the LabWindows/CVI DLL uses.
4. Click **OK** to close the Type Properties dialog box.
5. In the Types window, expand the `CVIStructExample` data type. You can right-click each property and select **Properties** from the context menu to launch the Type Properties dialog box for each property of the data type. You can review each property to see how it is configured.

Related concepts:

- [TestStand Directory Structure](#)
- [Programming with the TestStand API in LabWindows/CVI](#)
- [Sequence Local Variables](#)
- [Data Types](#)

Passing Structs to Code Modules - LabVIEW DLL

Purpose

This example demonstrates how to pass a TestStand container as a LabVIEW cluster to a LabVIEW VI. The container consists of several different data types. The VIs are compiled into a DLL and called as functions.



Note VIs always use a 1-byte packing scheme when passing clusters as parameters.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Passing Structs to  
Code Modules\LabVIEW DLL\Passing Structs to Code  
Modules.seq
```

Highlighted Features

- LabVIEW Adapter
- Local variables
- Data types

Major API

None

Prerequisites

You do not need to have the LabVIEW development system installed to use this example, but you must have the LabVIEW development system installed if you want to review the source VIs.

How to Use This Example

The sequence specifies `StructOut` and `StructIn` container local variables, which you can review on the Variables pane. When the sequence runs, TestStand copies the

values in the `StructOut` container to the individual local variables, and then copies the values again to the `StructIn` container. The LabVIEW DLL handles both operations. Before you run the sequence, the `StructOut` container contains values, the `StructIn` container contains default values, the individual local variables contains default values.

Complete the following steps to use this example.

1. Select **Execute»Test UUTs** to run the sequence.
2. Enter any serial number in the UUT Information dialog box and click **OK**. TestStand launches a Results for Unbundle dialog box that lists values loaded from the individual local variables, which specified default values before you executed the sequence.
3. Click **OK** in the Results for Unbundle dialog box. TestStand launches another Results for Unbundle dialog box that lists values loaded from the `StructIn` container, which specified default values before you executed the sequence.
4. Click **OK** when TestStand informs you that the sequence passed.
5. Click **Stop** in the UUT Information dialog box to end the execution.

You can review the two Message Popup steps in the sequence to verify that the appropriate sources are providing values. You can open `LV_DLL_1.vi` and `LV_DLL_2.vi`, located in the `<TestStand Public>\Examples\Fundamentals\Passing Structs to Code Modules\LabVIEW DLL` directory, to see how the data is being copied. The `LV_DLL_1.vi` uses an unbundle function to separate the values from the cluster, and the `LV_DLL_2.vi` uses a bundle function to combine individual values into a single cluster. The sequence is using `LVStructTest.dll`, into which `LV_DLL_1.vi` and `LV_DLL_2.vi` were built.

To pass a TestStand container as a struct to a DLL, you must define the container as a custom data type, and you must specify that objects of the data type can be passed as structs.

Complete the following steps to use the Types window to review the data type definition for a container.

1. In the Sequence File window, right-click the `StructIn` container on the Variables pane and select **View»Type Definition** from the context menu to launch the Types window.

2. Right-click the `DType1` data type and select **Properties** from the context menu to launch the Type Properties dialog box. Because the sequence is using a DLL, you must enable C struct passing for the data type.
3. Click the C Struct Passing tab. Notice that the Allow Objects of This Type to be Passed as Structs option is enabled. The Packing ring control specifies the 8 Byte Boundary option, which is the packing scheme the DLL uses.
4. Click **OK** to close the Type Properties dialog box.
5. In the Types window, expand the `DType1` data type. You can right-click each property and select **Properties** from the context menu to launch the Type Properties dialog box for each property of the data type. You can review each property to see how it is configured.

Related concepts:

- [TestStand Directory Structure](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Sequence Local Variables](#)
- [Data Types](#)

Passing Structs to Code Modules - .NET

Purpose

This example demonstrates how to pass a TestStand container as a .NET struct to a method in an assembly. This example uses functions to pass the struct by both reference and value. For a method in an assembly to change the value of a TestStand container, you must pass the reference of the container to the method.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Passing Structs to  
Code Modules\DotNet\Passing Structs to Code Modules.seq
```

Highlighted Features

- .NET Adapter
- Local variables
- Data types

Major API

None

Prerequisites

None

How to Use This Example

The sequence specifies `StructOut` and `StructIn` container local variables, which you can review on the Variables pane. The sequence uses a .NET method to copy the values from the `StructOut` container to the `StructIn` container first by reference and then by value.

Complete the following steps to use this example.

1. Select **Execute»Test UUTs** to run the sequence.
2. Enter any serial number in the UUT Information dialog box and click **OK**. TestStand launches a Get Results dialog box that lists values loaded from the `StructIn` container, which specified default values before you executed the sequence.
3. Click **OK** in the Get Results dialog box. TestStand launches another Get Results dialog box that lists values loaded from the `StructIn` container. The second `GetStructByRef` step specifies a pre-expression that specifies a value of 0 for each variable, and then the step specifies new values for each variable.
4. Click **OK** when TestStand informs you that the sequence passed.
5. Click **Stop** in the UUT Information dialog box to end the execution.

You can review the two Message Popup steps in the sequence to verify that the `StructIn` container is displaying the variable values. The methods the assembly is using are available in `StructPassingDotNet.dll`, located in the `<TestStand Public>\Examples\Fundamentals\Passing Structs to Code Modules\DotNet` directory.

Related concepts:

- [TestStand Directory Structure](#)
- [Sequence Local Variables](#)

- [Data Types](#)

Pausing Executions with Running Code Modules

The <TestStand Public>\Examples\Fundamentals\Pausing Executions with Running Code Modules directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Pausing Executions with Running Code Modules - LabWindows/CVI

Purpose

A TestStand execution cannot break until all threads within the execution have been suspended. If one or more threads in the execution is running a code module, the execution will not break until all code modules complete.

This example demonstrates two methods you can use to override this behavior:

- The Thread.ExternallySuspended property, which if set to True, allows an execution to be suspended even if the current thread is still executing a code module. You can use this option to configure the execution pausing behavior on a per code module basis.
- The Allow Break While in Code Modules station option, which allows any execution to be suspended even if one or more threads is still executing a code module. This setting is located in the Execution tab of the Station Options dialog. You can set this option at run-time, as shown in this example.

The example also demonstrates how you can poll the TestStand execution status within a code module and respond when the execution pauses.

Example File Location

<TestStand Public>\Examples\Fundamentals\Pausing Executions with Running Code Modules\CVI\Pausing Executions with

Running Code Modules.seq

Highlighted Features

Debugging TestStand Executions

Major API

Thread.ExternallySuspended

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. This example shows three configuration cases. For each case, the code module dialog indicates the current settings for the **Thread.ExternallySuspended** property and the **Allow Break While in Code Modules** station option.
3. In case one, observe that the TestStand execution does not pause when it reaches the break step. The code module also continues executing.
4. After you dismiss the dialog, the execution will pause. Click the **Resume** button or press F5 to resume the execution.
5. In cases two and three, the execution is able to pause successfully, and the code module pauses the simulated testing operation. If you resume the execution in TestStand while the module is still running, observe that the module resumes testing.

Related concepts:

- [TestStand Directory Structure](#)
- [Debugging Executions](#)

Pausing Executions with Running Code Modules - LabVIEW

Purpose

A TestStand execution cannot break until all threads within the execution have been suspended. If one or more threads in the execution is running a code module, the execution will not break until all code modules complete.

This example demonstrates two methods you can use to override this behavior:

- The `Thread.ExternallySuspended` property, which if set to `True`, allows an execution to be suspended even if the current thread is still executing a code module. You can use this option to configure the execution pausing behavior on a per code module basis.
- The `Allow Break While in Code Modules` station option, which allows any execution to be suspended even if one or more threads is still executing a code module. This setting is located in the `Execution` tab of the `Station Options` dialog. You can set this option at run-time, as shown in this example.

The example also demonstrates how you can poll the TestStand execution status within a code module and respond when the execution pauses.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Pausing Executions  
with Running Code Modules\LabVIEW\Pausing Executions with  
Running Code Modules.seq
```

Highlighted Features

Debugging TestStand Executions

Major API

`Thread.ExternallySuspended`

Prerequisites

You must have the LabVIEW development system installed and configure the LabVIEW Adapter to use the LabVIEW development system.

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. This example shows three configuration cases. For each case, the code module dialog indicates the current settings for the **Thread.ExternallySuspended** property and the **Allow Break While in Code Modules** station option.
3. In case one, observe that the TestStand execution does not pause when it reaches the break step. The code module also continues executing.
4. After you dismiss the dialog, the execution will pause. Click the **Resume** button or press F5 to resume the execution.
5. In cases two and three, the execution is able to pause successfully, and the code module pauses the simulated testing operation. If you resume the execution in TestStand while the module is still running, observe that the module resumes testing.

Related concepts:

- [TestStand Directory Structure](#)
- [Debugging Executions](#)

Pausing Executions with Running Code Modules - .NET

Purpose

A TestStand execution cannot break until all threads within the execution have been suspended. If one or more threads in the execution is running a code module, the execution will not break until all code modules complete.

This example demonstrates two methods you can use to override this behavior:

- The **Thread.ExternallySuspended** property, which if set to True, allows an

execution to be suspended even if the current thread is still executing a code module. You can use this option to configure the execution pausing behavior on a per code module basis.

- The Allow Break While in Code Modules station option, which allows any execution to be suspended even if one or more threads is still executing a code module. This setting is located in the Execution tab of the Station Options dialog. You can set this option at run-time, as shown in this example.

The example also demonstrates how you can poll the TestStand execution status within a code module and respond when the execution pauses.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Pausing Executions  
with Running Code Modules\DotNet\Pausing Executions with  
Running Code Modules.seq
```

Highlighted Features

Debugging TestStand Executions

Major API

Thread.ExternallySuspended

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. This example shows three configuration cases. For each case, the code module dialog indicates the current settings for the **Thread.ExternallySuspended** property and the **Allow Break While in Code Modules** station option.
3. In case one, observe that the TestStand execution does not pause when it reaches

the break step. The code module also continues executing.

4. After you dismiss the dialog, the execution will pause. Click the **Resume** button or press F5 to resume the execution.
5. In cases two and three, the execution is able to pause successfully, and the code module pauses the simulated testing operation. If you resume the execution in TestStand while the module is still running, observe that the module resumes testing.

Related concepts:

- [TestStand Directory Structure](#)
- [Debugging Executions](#)

Interpreter Session Management in Python

Purpose

The example will showcase how to use different Python interpreter sessions in a single execution.

Example File Location

```
<TestStand Public>Examples\Fundamentals\Python\Interpreter  
Session Management\Interpreter Session Management  
Sequence.seq
```

Highlighted Features

- Python Interpreter Session Management

Major API

None

Prerequisites

You must have required version of CPython interpreter installed and added to your PATH variable. You must update the Python version to use in the Python adapter

configuration dialog before using the example.

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane of the MainSequence sequence, review the steps in the Setup step group.
 - The first step, Update Python version to use in all steps, updates the Python version in each step to use the version specified in Python adapter settings. You typically will not do this for an actual test and should instead directly specify the Python version to use in the steps that use the Python adapter. This example can't determine the Python version installed in the computer and therefore modifies the Python version of each step programmatically to match what is specified in the Python adapter settings.
2. On the Steps pane, review the steps in the Main step group.
 - The Global interpreter session step, which is a Sequence Call step, call the Global subsequence two times to demonstrate Python global interpreter session.
 - The Per Thread interpreter session, which is a Sequence Call step, call the Per Thread subsequence three times on a separate thread to demonstrate Python per thread interpreter session.
 - The Wait for Thread Return step, which is a Wait step, waits for execution completion of the Per Thread subsequence on a separate thread.
 - The Per Execution interpreter session, which is a Sequence Call step, call the Per Execution subsequence on a new execution instance to demonstrate Python per execution interpreter session.
 - The Wait for Executions to Return step, which is a Wait step, waits for execution completion of the Per Execution subsequence on a separate execution instance.
 - The Object Reference interpreter session step, which is a Sequence Call step, call the Object Reference subsequence to demonstrate Python object reference interpreter session.
3. The Global, Per Thread, Per Execution, and Object Reference subsequence gets and sets Python attribute in the code module.
4. The Result step in Global, Per Thread, Per Execution, and Object Reference subsequence, which is an Additional Results step, summarizes the results in the report.

5. Select **Execute»Single Pass** to run the sequence.
6. When execution completes, review the report to see the how different interpreter sessions maintain independence from each other.

Related concepts:

- [TestStand Directory Structure](#)
- [Step Groups](#)

Passing Data between TestStand and Python

Purpose

This example demonstrates how you can pass TestStand data types to Python code modules.

Example File Location

```
<TestStand Public>Examples\Fundamentals\Python\Passing Data  
to Python\Passing Data to Python.seq
```

Highlighted Features

- Mapping of Python and TestStand types

Major API

None

Prerequisites

Complete the following steps to install the required software and configure TestStand.

1. Install a supported version of the CPython interpreter and add it to your PATH variable.
2. Install the following libraries for the version of Python you installed:
 - Python for Win32 (pywin32) extensions
 - NumPy module (required for the NumPy Array example)
3. In the Python Adapter Configuration dialog box, update the Python version

TestStand uses to execute Python code modules.



Note To run the example for Enums, you must use Python 3.6 or higher.

How to Use This Example



Note The example simulates the testing of a UUT. The Model type, serial number, ports and voltage values in the example are hypothetical. The methods in the classes `UUTInterface` and `UUTDatabase` in the Python module are template methods and do not interface with any hardware or database.

Review the sequences and steps in this example. Select a subsequence in the Sequences pane to view steps in the subsequence.

Sequence Call step in the MainSequence sequence	Subsequence the step calls	Purpose	Details
Working with Number	Number	Demonstrates passing numbers (integers and real) between TestStand and the Python module.	The subsequence has four sections that demonstrate adding, subtracting, multiplying, and dividing two TestStand numbers in the Python module. The four sections, delineated by Label steps, use 64-bit signed integers, 64-bit unsigned integers, 64-bit double-precision floating-point number, and float and integer together respectively. An Additional Results step completes each section to provide a summary of the results.
Working with Boolean	Boolean	Demonstrates passing Boolean data between TestStand and the Python module.	The subsequence demonstrates the inversion of the Boolean state of a TestStand variable in a Python module. An Additional Results step provides a summary of the results.
Working with String	String	Demonstrates passing string data between	The subsequence demonstrates the concatenation of two TestStand strings in a Python module. An Additional Results step provides a summary of the

Sequence Call step in the MainSequence sequence	Subsequence the step calls	Purpose	Details
		TestStand and the Python module.	results.
Working with List	List	Demonstrates passing data between TestStand arrays and Python lists.	The subsequence demonstrates different operations, such as sorting and converting array data to string data, the Python module performs on TestStand array data.
Working with Tuple	Tuple	Demonstrates passing data between TestStand containers and Python tuples.	The subsequence retrieves a single element from a TestStand container using a Python module, sets a Python attribute as a tuple using a TestStand container, and inserts a Python tuple attribute into a TestStand container.
Working with PyObject	PyObject	Demonstrates passing data between TestStand object references and Python objects (PyObject).	The Setup step group contains the following steps: • The Create UUTInterface Object step and the Initialize Database step obtain the references to the instances of the Python classes that are instantiated in the Python module. TestStand stores the references in TestStand object reference variables. Steps in the Main and Cleanup step groups will use these references. The Main step group contains the following steps: • The Get UUT Model Type step, which is an action step, gets the value in the static attribute of the Python class <code>UUTInterface</code>. • The Get UUT Serial Number step, which is an action step, gets the value of the attribute <code>serialNumber</code> from the object of the class <code>UUTInterface</code>, whose reference was obtained earlier.

Sequence Call step in the MainSequence sequence	Subsequence the step calls	Purpose	Details
			- The Create UUT Record in Database step, which is an action step, calls the <code>createRecord</code> method in the class instance object of Python class <code>UUTDatabase</code>, whose reference was obtained earlier. - The Powerup UUT step, which is an action step, calls the <code>PowerupUUT</code> method in the class instance object of the Python class <code>UUTInterface</code> whose reference was obtained earlier. - The UUT String Value Test step validates the serial number of the <code>UUTInterface</code> object. - The UUT Multiple Numeric Limit Test step validates the range of ports assigned to the <code>UUTInterface</code> object. - The UUT Numeric Limit Test step validates the voltage read by the <code>UUTInterface</code> object. - The UUT Pass/Fail Test step validates the test result of the <code>UUTInterface</code> object.
Working with Dispatch	Dispatch	Demonstrates passing TestStand objects and COM objects implementing <code>IDispatch</code> between TestStand and the Python module.	The Main step group contains the following Action steps: - The Pass Sequence Context as Parameter step passes the <code>SequenceContext</code> object as a parameter to the Python code module. - The Pass Engine as Parameter step passes the <code>Engine</code> property as a parameter to the Python code module. - The Pass COM Object as Parameter step passes the COM object, whose reference is held in an object reference variable. The Pre-Expression of the step makes the object reference variable hold a reference to a <code>PropertyObject</code>, which is a COM object.

Sequence Call step in the MainSequence sequence	Subsequence the step calls	Purpose	Details
			When the step uses the object reference variable as a parameter, it passes the COM object held by the variable to the Python code module.
Working with Enum	Enum	Demonstrates passing enumerations between TestStand and the Python module.	The Main step group contains the following Action steps: • The Pass Enum Color step passes the TestStand enum as a Python enum parameter with a type of <code>Enum</code> to the Python code module. • The Pass Flag Measurement step passes the TestStand enum as a Python enum parameter with a type of <code>Flag</code> to the Python code module. • The Pass IntEnum FileOpenOption step passes the TestStand enum as a Python enum parameter with a type of <code>IntEnum</code> to the Python code module. • The Pass IntFlag FilePermission step passes the TestStand enum as a Python enum parameter with a type of <code>IntFlag</code> to the Python code module.
Working with NumPy Array	NumPy Array	Demonstrates passing data between a TestStand array and a Python NumPy array.	The subsequence demonstrates passing TestStand arrays to a Python module and performing a sorting operation using NumPy arrays.

Complete the following steps to run the example.

1. Click **Execute»Single Pass** to run the sequence.

2. When execution completes, review the report.

Related concepts:

- [TestStand Directory Structure](#)

Sequence File Translators

Purpose

This example demonstrates how to create and use a sequence file translators, which is a DLL that TestStand uses to load sequence files in custom formats. A translator implements a set of required callbacks that TestStand invokes to determine the types of files the translator can open and to load a custom file as a TestStand sequence file. You can use translators to open files in custom formats. Once TestStand loads a custom file as a TestStand sequence file, you can operate on the file like any other TestStand sequence file. However, you cannot save changes in the TestStand sequence file back to the original format.

The example directory contains projects for example translators written in LabVIEW, LabWindows/CVI, and Microsoft Visual C++. The examples illustrate translating text and XML files and contain a template project you can use to develop a translator in each development environment. Each example can translate the corresponding text or XML file located in the same directory as the example.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Sequence File  
Translators - Examples\SetupExampleTranslators.seq
```

Highlighted Features

Sequence file translators

Major API

N/A

Prerequisites

Before you attempt to open a file in a custom format, you must install the example translator DLL and support files to the appropriate directories in one of the following ways:

- **Automatic Installation (recommended)**—Open the `SetupExampleTranslators.seq` example sequence file and run the `MainSequence` to select the examples to set up.
- **Manual Installation**—Complete the following steps to manually move the translator DLL and support files:
 1. Open the `TextTranslator` or `XMLTranslator` directory for one of the examples in the `<TestStand Public>\Examples\Fundamentals\Sequence File Translators - Examples` directory, and copy the translator DLL from the selected directory to the `<TestStand Public>\Components\Translators` directory.
 2. Copy the `NI_ExampleTranslatorTypes.ini` type file from the `<TestStand Public>\Examples\Fundamentals\Sequence File Translators - Examples` directory to the `<TestStand Public>\Components\TypePalettes` directory.
 3. Copy the `FrontPanel.uir` and `StepType.dll` files from the `<TestStand Public>\Examples\Fundamentals\Sequence File Translators - Examples\Common` directory to the `<TestStand Public>\Components\StepTypes\Translators` directory.

You must restart the TestStand Sequence Editor so that TestStand can load the example translator DLL.

How to Use This Example

To open a file with a custom format, select **File»Open** to launch the Open File dialog box. The Files of Type control in the Open File dialog box contains the new file extensions the translator supports. Browse to the examples directory that corresponds to the development environment you are using and the custom format for which you installed a translator and select `SampleFile`.



Note If you modify and rebuild an example translator DLL, you must copy

the DLL to the <TestStand Public>\Components\Translators directory and restart TestStand to load the new DLL.

Related concepts:

- [Sequence File Translators](#)
- [TestStand Directory Structure](#)

Structure of TestStand Executions

Purpose

This example demonstrates the concepts of executions, threads, and call stacks and shows how they interact. The example also shows how you can use the Call Stack pane and the Threads pane to view the current status of an execution.

Example File Location

<TestStand Public>\Examples\Fundamentals\Structure of TestStand Executions\Structure of TestStand Executions.seq

Highlighted Features

- Sequence Call step type
- Call Stack pane
- Threads pane

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequences and steps in the example.

1. On the Sequences pane, select the **MainSequence**. This sequence contains three Sequence Call steps, which represent three distinct methods of calling a subsequence.
2. Click the **Call SubSequence** step and navigate to the Module tab of the Step Settings pane. Notice that the Execution Options for this sequence call are set to None, which indicates that this sequence does not execute in a new thread or execution.
3. Click the **Call NewThread** step and navigate to the Execution Options of this step. Notice that this sequence call is set to execute in a new thread, which means that execution of the MainSequence continues as the New Thread sequence executes.
4. Click the **Call NewExecution** step and navigate to the Execution Options of this step. Notice that this sequence call is set to execute in a new TestStand execution, which means that TestStand manages execution of the NewExecution sequence independently from the current execution.

Complete the following steps to run the example.

1. Select **Execute»Single Pass** to run the sequence.
2. During execution, message prompts explain the structure of TestStand executions. After you dismiss each message, the corresponding Sequence Call step executes, and TestStand automatically pauses execution so you can examine each concept being demonstrated.
3. When you are ready to continue to the next step in the example, click the green **Resume** button on the Debug toolbar. This process repeats until execution completes.

Related concepts:

- [TestStand Directory Structure](#)

Termination Monitor

The <TestStand Public>\Examples\Fundamentals\Termination Monitor directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Termination Monitor - LabWindows/CVI

Purpose

This example demonstrates the use of the Termination Monitor function in LabWindows/CVI. The Termination Monitor allows a code module to stop executing if TestStand attempts to terminate or abort the execution.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Termination  
Monitor\CVI\Termination Monitor.seq
```

Highlighted Features

- LabWindows/CVI Adapter
- Termination Monitor

Major API

None

Prerequisites

None

How to Use This Example

This example uses a CVI Action step to call a function in a C DLL that runs indefinitely. Since it uses the Termination Monitor functionality, the function is able to stop when TestStand attempts to terminate or abort the execution. If the function did not call the Termination Monitor method, TestStand would be unable to terminate or abort the execution.

To review the functions used in this example, open `Termination Monitor Example.c` located at `<TestStand Public>\Examples\Fundamentals\Termination Monitor\CVI\Termination Monitor Example.c`.

The `TS_CancelDialogIfExecutionStops` method is used to search for a request from TestStand to stop the execution. This method automatically initiates a timer callback, which will call the `QuitUserInterface` method if TestStand is attempting to terminate or abort the execution.

Note that the `TS_CancelDialogIfExecutionStops` method requires the panel handle of the onscreen dialog and the `SequenceContext` of the calling sequence as parameters. This method also assumes that the calling code has control of the dialog through a call to the `RunUserInterface` method.

Complete the following steps to run the example:

1. In TestStand, select **Execute»Single Pass** to run the sequence.
2. When the `TerminationMonitorExample` dialog appears, the sequence will execute indefinitely.
3. On the Debug toolbar in TestStand, click the red **Terminate** button to terminate the execution. Observe that the execution terminates successfully.

Related concepts:

- [TestStand Directory Structure](#)
- [Programming with the TestStand API in LabWindows/CVI](#)
- [Checking for Suspended or Stopped Execution within Code Modules](#)

Termination Monitor - LabVIEW

Purpose

This example demonstrates the use of the Termination Monitor VIs in LabVIEW. The Termination Monitor allows a code module to stop executing if TestStand attempts to terminate or abort the execution.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Termination
Monitor\LabVIEW\Termination Monitor.seq
```

Highlighted Features

- LabVIEW Adapter
- Termination Monitor

Major API

None

Prerequisites

To run this example, you must have the LabVIEW development system installed and you must configure the LabVIEW Adapter to use the LabVIEW development system.



Note The Termination Monitor palette VIs do not require the LabVIEW development system. This requirement applies only to this example.

How to Use This Example

This example uses a LabVIEW Action step to call a VI that runs indefinitely. Since this VI uses the Termination Monitor functions, it is able to stop when TestStand attempts to terminate or abort the execution. If the VI did not include the Termination Monitor functions, TestStand would be unable to terminate or abort the execution.

To review the configuration of the VI, open `Termination Monitor Example.vi` located at `<TestStand Public>\Examples\Fundamentals\Termination Monitor\LabVIEW\Termination Monitor Example.vi`. In the VI, observe that the TestStand - Initialize Termination Monitor VI is used to create a reference to the Termination Monitor object.

Every 100 milliseconds, the TestStand - Get Termination Monitor Status VI is called to determine whether TestStand is attempting to stop the execution. If so, the VI exits.

Note that the Termination Monitor VIs require the SequenceContext of the calling sequence. The SequenceContext is passed to the code module as a parameter.

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to run the sequence.
2. When the TerminationMonitorExample VI front panel appears, the sequence will execute indefinitely.
3. On the Debug toolbar in TestStand, click the red **Terminate** button to terminate the execution. Observe that the execution terminates successfully.

Related concepts:

- [TestStand Directory Structure](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Checking for Suspended or Stopped Execution within Code Modules](#)

Termination Monitor - .NET

Purpose

This example demonstrates the use of the Termination Monitor function in .NET. The Termination Monitor allows a code module to stop executing if TestStand attempts to terminate or abort the execution.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Termination  
Monitor\dotNET\Termination Monitor.seq
```

Highlighted Features

- .NET Adapter
- Termination Monitor

Major API

None

Prerequisites

None

How to Use This Example

This example uses a .NET Action step to call a function in a .NET assembly that runs indefinitely. Since it uses the Termination Monitor functionality, the function is able to stop when TestStand attempts to terminate or abort execution. If the function did not call the Termination Monitor method, TestStand would be unable to terminate or abort the execution.

To review the functions used in this example, open `Termination Monitor Example Form.cs` located at `<TestStand Public>\Examples\Fundamentals\Termination Monitor\dotNET\Termination Monitor Example\Termination Monitor Example Form.cs`.

In the `InitializeTerminationStateChecking()` method, notice that the `Execution.InitTerminationMonitor` method from the TestStand API is used to initialize the termination monitor and obtain a reference to data required by other Termination Monitor functions. Every 100 milliseconds, the state of the execution is queried with the `Execution.GetTerminationMonitorStatus` method from the TestStand API. If TestStand is attempting to terminate or abort the execution, this code module finishes its execution.

Complete the following steps to run the example:

1. In TestStand, select **Execute»Single Pass** to run the sequence.
2. When the `TerminationMonitorExample` dialog appears, the sequence will execute indefinitely.
3. On the Debug toolbar in TestStand, click the red **Terminate** button to terminate the execution. Observe that the execution terminates successfully.

Related concepts:

- [TestStand Directory Structure](#)
- [Checking for Suspended or Stopped Execution within Code Modules](#)

Using Data Streams

The `<TestStand Public>\Examples\Fundamentals\Using Data`

Streams directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Using Data Streams - LabVIEW

Purpose

This example uses data streams to read from a comma separated value (csv) file and to create a csv file.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Using Data  
Streams\LabVIEW\UsingDataStreams.seq
```

Highlighted Features

- TestStand Data Streams

Major API

- TestStand Data Streams API

Prerequisites

You must have the LabVIEW Development System installed to execute this example.

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. View the resulting CSV in the Report tab.

Related concepts:

- [TestStand Directory Structure](#)

Using Data Streams - .NET

Purpose

This example uses data streams to read from a comma separated value (csv) file and to create a csv file.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Enumerations\  
UsingDataStreams.seq
```

Highlighted Features

- TestStand Data Streams

Major API

- TestStand Data Streams API

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. View the resulting CSV in the Report tab.

Related concepts:

- [TestStand Directory Structure](#)

Using Sweep Loops

The <TestStand Public>\Examples\Fundamentals\Using Sweep Loops directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Using Sweep Loops - LabVIEW

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Using Sweep Loops\  
LabVIEW\Using Sweep Loops.seq
```

Highlighted Features

- Sweep Loops
- Data Streams

Major API

- TestStand Data Streams API

Prerequisites

LabVIEW development system 16.0 or later

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run Mainsequence** to execute the sequence.

Related concepts:

- [TestStand Directory Structure](#)

Using Sweep Loops - .NET

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Using Sweep Loops\  
dotNet\Using Sweep Loops.seq
```

Highlighted Features

- Sweep Loops
- Data Streams

Major API

- TestStand Data Streams API

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run Mainsequence** to execute the sequence.

Related concepts:

- [TestStand Directory Structure](#)

Sweep Loop Strategies

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Using Sweep Loops\  
Sweep Loop Strategies Demo\Sweep Loop Strategies.seq
```


Highlighted Features

- Sweep Loops

Major API

- SetBreakSettings

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run Mainsequence** to execute the sequence
2. When the first breakpoint is hit, ensure that `Locals.SweepLoopParameterValues` is added to the Watch Window during execution.
3. At each breakpoint, observe the difference in the sweep loop strategies. The strategy for `ConstantValue` resets the variables value to the value set in the parameters list of the sweep loop even though it is incremented during an iteration of the loop. The strategy for `CaptureValue` retains the new value when a new iteration of the loop starts.

Related concepts:

- [TestStand Directory Structure](#)

Using TestStand Enumerations

The `<TestStand Public>\Examples\Fundamentals\Using TestStand Enumerations` directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Using TestStand Enumerations - CVI

Purpose

This example shows how to use TestStand enumerations with CVI code modules. Enumerations are defined as custom types and can be stored in the Sequence File or Type Palettes.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Using TestStand Enumerations\CVI\Using TestStand Enumerations.seq
```

Highlighted Features

- Using Enumerations in TestStand
- Range Checking: Strict and Loose Enumerations
- Flags Enumerations and Bitwise OR Semantics
- Using Enumerations with the TestStand Adapters

Major API

- Val Function
- Str Function

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. As the example executes, follow the prompts and read the message popups for information about Enumerations, Strict Enumerations, and Flag Enumerations.

Complete the following steps to review the enumerations and steps in this example:

1. Open the **Enum** sequence from the Sequences pane. Select the **Enum Behavior** step and review the step settings and parameter values for the step. This code module allows the user to select a combination of RGB colors, which are then stored in the `Locals.Color` Local variable.
2. Select the **Display Enum Value** step and review the Message Expression. This expression uses the `Str()` and `Val()` functions to display the string and formatted numeric representations of an enum value.
3. Repeat steps 1 and 2 for the **Strict Enum** and **Flag Enum** sequences.
4. Open the Types Window (Ctrl + t) and select `Enums.seq` to view the three different enumeration definitions used in this example: `Colors`, `Colors_Strict`, and `Colors_Flag`. Choose **Edit Enumerations** to view details of each type.

Related concepts:

- [TestStand Directory Structure](#)
- [Using Enumerations in TestStand](#)
- [Range Checking: Strict and Loose Enumerations](#)
- [Flags Enumerations and Bitwise OR Semantics](#)
- [Using Enumerations with the TestStand Adapters](#)

Using TestStand Enumerations - LabVIEW

Purpose

This example shows how to use TestStand enumerations with LabVIEW code modules. Enumerations are defined as custom types and can be stored in the Sequence File or Type Palettes.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Using TestStand Enumerations\LabVIEW\Using TestStand Enumerations.seq
```

Highlighted Features

- Using Enumerations in TestStand
- Range Checking: Strict and Loose Enumerations

- Flags Enumerations and Bitwise OR Semantics
- Using Enumerations with the TestStand Adapters

Major API

- `PropertyObject.GetPropertyObject`
- `SequenceFile.AsPropertyObjectFile`
- `SequenceFile.AsPropertyObject`

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. As the example executes, follow the prompts and read the message popups for information about Enumerations.

Complete the following steps to review the enumerations and steps in this example:

1. Open the **Enum** sequence from the Sequences pane. Select the **Enum Behavior** step and review the step settings and parameter values for the step. This code module allows the user to select a combination of RGB colors, which are then stored in the `Locals.Color` Local variable.
2. Select the **Display Enum Value** step and review the Message Expression. This expression uses the `Str()` and `Val()` functions to display the string and formatted numeric representations of an enum value.
3. Repeat steps 1 and 2 for the **Strict Enum** and **Flag Enum** sequences.
4. Open the Types Window (Ctrl + t) and select `Enums.seq` to view the three different enumeration definitions used in this example: `Colors`, `Colors_Strict`, and `Colors_Flag`. Choose **Edit Enumerations** to view details of each type.

Related concepts:

- [TestStand Directory Structure](#)
- [Using Enumerations in TestStand](#)
- [Range Checking: Strict and Loose Enumerations](#)
- [Flags Enumerations and Bitwise OR Semantics](#)
- [Using Enumerations with the TestStand Adapters](#)

Using TestStand Enumerations - .NET

Purpose

This example shows how to use TestStand enumerations with .NET code modules. Enumerations are defined as custom types and can be stored in the Sequence File or Type Palettes.

Example File Location

```
<TestStand Public>\Examples\Fundamentals\Using TestStand  
Enumerations\DotNet\Using TestStand Enumerations.seq
```

Highlighted Features

- Using Enumerations in TestStand
- Range Checking: Strict and Loose Enumerations
- Flags Enumerations and Bitwise OR Semantics
- Using Enumerations with the TestStand Adapters

Major API

- Val Function
- Str Function

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. As the example executes, follow the prompts and read the message popups for information about Enumerations, Strict Enumerations, and Flag Enumerations.

Complete the following steps to review the enumerations and steps in this example:

1. Open the **Enum** sequence from the Sequences pane. Select the **Enum Behavior** step and review the step settings and parameter values for the step. This code module allows the user to select a combination of RGB colors, which are then stored in the `Locals.Color` Local variable.
2. Select the **Display Enum Value** step and review the Message Expression. This expression uses the `Str()` and `Val()` functions to display the string and formatted numeric representations of an enum value.
3. Repeat steps 1 and 2 for the **Strict Enum** and **Flag Enum** sequences.
4. Open the Types Window (Ctrl + t) and select `Enums.seq` to view the three different enumeration definitions used in this example: `Colors`, `Colors_Strict`, and `Colors_Flag`. Choose **Edit Enumerations** to view details of each type.

Related concepts:

- [TestStand Directory Structure](#)
- [Using Enumerations in TestStand](#)
- [Range Checking: Strict and Loose Enumerations](#)
- [Flags Enumerations and Bitwise OR Semantics](#)
- [Using Enumerations with the TestStand Adapters](#)

Examples for Interfacing with Hardware

The Interfacing with Hardware examples demonstrate how you can use TestStand to communicate with and coordinate your test hardware resources. This directory includes the following example programs.

Instrument Control Step Types

Purpose

This example demonstrates how to implement a TestStand HP34401a instrument control step type in LabWindows/CVI. You can use the step type to access commonly-used functionality of the HP34401a multimeter from a TestStand sequence.

Example File Location

```
<TestStand Public>\Examples\Interfacing with Hardware\  
Instrument Control Step Types - HP34401a\Instrument Control  
Step Types - HP34401a.seq
```

Highlighted Features

TestStand step types

Major API

N/A

Prerequisites

You must install the HP34401a IVI instrument driver from the Instrument Driver Network at ni.com/devzone/idnet before running this example.



Note Depending on the version and configuration of IVI on the computer, simulated measurements the HP34401a step type makes might not return realistic values. The measurements are correct if you use a real HP34401a instrument.

How to Use This Example

This example demonstrates optional features you can include in step types you create. You can decide which, if any, of the following features to support in step types you create:

- Tabbed editing dialog box
- Multiple selectable run-time operations
- Configuration settings you can specify with TestStand expressions
- Groups of configuration settings you can load or save to a TestStand custom data type, which you can then execute according to a group of settings you store in a local variable or receive as a sequence parameter
- Instrument driver session obtained from the Session Manager to maximize the sharing of the instrument handle between steps and other code modules
- Multiple language string support

Installation Issues

Step types might depend on files such as code modules, icon files, or string resource files. For TestStand to locate these files, the files must reside in specific directories.

- **Code Module Files (.dll, .vi, and so on)**—Place step type code module files in a subdirectory you create in the `<TestStand Public>\Components\StepTypes` directory. TestStand finds code modules you place in this directory because the TestStand default search paths include all directories under the `<TestStand Public>\Components` directory. Although this example does not install its code module, `HP34401aStepType.dll`, to the `<TestStand Public>\Components` directory, TestStand can find the step type code module because the step type specifies a relative path from the `<TestStand Public>\Examples\Interfacing with Hardware\Instrument Control Step Types - HP34401a` directory.
- **Icon Files**—Place icon files for step types in the `<TestStand Public>\Components\Icons` directory. TestStand loads icons from this directory and the `<TestStand>\Components\Icons` directory. TestStand installs the icon for this example to `<TestStand Public>\Components\Icons\NI_Examples\Example_HP34401a.ico`.
- **String Resource Files**—Place string resource files you create in the `<TestStand Public>\Components\Language\<language>\TestStandExamples.ini` file.

The `HP34401a_StepTypeExample.seq` example sequence file contains the example step type definition. You can store step type definitions you create in any sequence file, and you can add a step type to a new or existing TestStand type palette

file. You can install a type palette file on another computer by placing the file in the `<TestStand Public>\Components\TypePalettes` directory. You must prefix the filenames of the type palettes you install with `Install_`. At startup, TestStand searches the `TypePalettes` directory for type palette files with the `Install_` prefix. When TestStand finds a palette file to install and the base filename is not the same as any existing palette file, TestStand removes the `Install_` prefix and adds the palette to the palette list. When TestStand finds a palette file to install and the base filename is the same as an existing palette file, TestStand merges the types from the `Install_` prefixed file with the existing palette file and deletes the `Install_` file.

Step Type Implementation

Complete the following steps to implement a step type.

1. Create a new step type in the Types window. The example step type is named `HP34401aStepType`, which you can view in the Types window for the `Instrument Control Step Types - HP34401a.seq` file.
2. Determine the data requirements for the step. Analyze the data inputs and outputs the step requires at run time.
3. Create input properties. Create a step subproperty for each data input. Select the appropriate data type and set the default value. The example step type uses the following data input properties:
`Step.LogicalNameStep.OperationStep.SettingSourceStep.Configuration`
4. Create output properties. Create a step subproperty for each data output. If you want an output property to automatically appear in the sequence result list, create the output property under the `Step.Result` property. The example step type uses the following data output properties:
`Step.Read.DestStep.SelfTest.DestStep.Result.SinglePointStep.Result`
5. Designate an adapter, if appropriate. If the step is not designed to call user-written code, enable the **Designate an Adapter** option on the General tab of the Step Type Properties dialog box and select the `<None>` Adapter. The example step type does not require you to write any code to access the instrument. Thus, the example step type designates the `<None>` Adapter as the appropriate module adapter.
6. Define run-time functionality. Create a Pre- or Post-Step substep to call a code module you write. Each instance of the step type calls the code module when the step executes. The code module call is transparent to the user. If the step type calls

user-written code, define a Pre- or Post-Step substep, depending on whether you want to call the code module before or after the step executes. If the step type does not call user-written code, you can use a Pre- or Post-Step substep to call the code module. The example step type defines a Post-Step substep that calls the `ExecuteHP34401aStep` DLL function when an instance of the step type executes. The `ExecuteHP34401aStep` DLL function accepts a **SequenceContext** parameter, which the function uses with the TestStand API to access the properties of the step. You can also define the function to accept a separate parameter for each step property the function uses.

7. Define an editing dialog box. Configure the Edit substep to call a code module that launches the dialog box, which shows step property values in a controlled and organized manner. When the user finishes editing, the dialog box stores the edited property values in the step and marks the sequence file that contains the step as modified. The Edit substep for the example step type calls the `EditHP34401aStep` DLL function to launch a dialog box in which the user selects the action the step executes at run time and configures the parameters the action requires. The `ExecuteHP34401aStep` DLL function accepts a **SequenceContext** parameter, which the function uses with the TestStand API to access the properties of the step. You can also define the function to accept a separate parameter for each step property the function uses.
8. Configure built-in step type properties. Use the Step Type Properties dialog box to configure the default values of built-in step type properties and other settings. Typically, you configure the following settings for a step type:
 - **Designate an Icon**—If the step does not call user-written code, select an icon that represents its functionality. You can select from the icon files in the `<TestStand Public>\Components\Icons` and `<TestStand>\Components\Icons` directories. If the step calls user-written code, do not designate an icon. The step displays the icon that represents the adapter that calls the user-written code. The example step type specifies `<TestStand Public>\Components\Icons\NI_Examples\Example_HP34401a.ico` as its icon file.
 - **Default Step Name Expression**—The default step name expression determines the name the sequence editor assigns to a new instance of the step type. You can use the `ResStr` expression function in the default name expression to specify a name that changes depending on the selected language. The example step type specifies the following expression as its default step name expression: `ResStr("HP34401_STEP_TYPE_EXAMPLE",`

"DEFAULT_NAME") If the step type does not need to account for the selected language, you can enter a default step name in quotes, such as "HP34401a Step".

- **Step Description Expression**—The step description expression determines the description that Description column on the Steps pane and the Description control on the General panel of the Properties tab of the Step Settings pane show for the step. The description expression can refer to and show the values of step properties. You can use the `ResStr` expression function in the description expression to specify a description that changes depending on the selected language. The example step type uses a lengthy expression to vary the description depending on the step configuration and the selected language. If the step type description does not need to account for the step configuration or the selected language, you can enter a fixed description in quotes, such as "HP33401a DMM Instrument Control".
- **Menu Item Name**—Use the **Item Name Expression** control on the Menu tab of the Step Type Properties dialog box to specify the item name expression for the step type. The expression determines the name of the menu item in the Insert Step submenu that inserts an instance of the step type into a sequence. You can use the `ResStr` expression function in the item name expression to specify an item name that changes depending on the selected language. The example step type specifies the following item name expression:
`ResStr("HP34401_STEP_TYPE_EXAMPLE", "MENU_NAME")` If the menu item does not need to account for the selected language, you can enter an item name in quotes, such as "HP34401a".
- **Other Items**—Depending on the step type, you can also use the following features. The example step type does not configure any other step type settings.
 - Configure the default values of built-in properties, such as the run options or loop options.
 - Use the Disable Properties tab of the Step Type Properties dialog box to prevent users of the step type from editing the values of built-in properties you select.
 - Specify code templates to assist the user in creating code modules that you designed the step type to call.

Suggestions for Implementing Instrument Control Step Types

To increase performance, do not initialize and close the instrument driver each time the step executes. Instead, use DLL global variables or another method to maintain the instrument instance between steps.

One option is to use the Session Manager to initialize and share instrument driver handles between multiple instances of the step type. The example step type obtains an instrument session from the Session Manager and attaches the instrument session to the current TestStand execution object by dynamically creating an ActiveX reference subproperty. In this way, the driver for the session stays open for the duration of the execution and all steps that can run during the execution share the same driver instance handle. You can also use the Session Manager to obtain a list of available instrument connections from which the step user can select.

To increase the flexibility of the step type, allow the user to specify the values of important settings with an expression. The user can then store the setting value in a variable and alter it at run time. The example step type allows the user to specify almost all setting values with an expression.

For groups of related settings, consider allowing the user to load or store the values in a variable of a custom data type that you define. The user can then specify a group of settings in a variable and pass the variable as a parameter to subsequences that contain the steps that apply the setting values. The example step type allows the user to specify a setting source variable to which the step loads or stores the settings for the selected operation.

Suggestions for Implementing Step Type Editing Dialog Boxes in LabWindows/CVI

The `<TestStand>\API\CVI\TSUtil.fp` instrument driver contains the following functions to help create a step type editing dialog box:

- If the text labels that appear on the step type editing dialog box must appear in the selected language, call the `TS_LoadPanelResourceStrings` function to load the language-specific dialog strings.
- Call the `TS_StartModalDialog` and `TS_EndModalDialog` functions to ensure that the editing dialog box is modal to the sequence editing application.
- Call the `TS_ExprCtrl_Create` function to convert a string control into an

expression editing control.

- Call the `TS_ExchangePropertyAndCtrlVals` function to transfer values between step properties and dialog box controls.
- Call the `TS_IncSequenceFileChangeCount` function if the editing dialog box modifies the step.

Related concepts:

- [TestStand Directory Structure](#)
- [Built-In Step Types](#)
- [Creating Custom Step Types](#)

InstrumentStudio

The `<TestStand Public>\Examples\Interfacing with Hardware\InstrumentStudio` directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Measure Efficiency

Purpose

This example demonstrates how to use IO Configuration step types to communicate with the instrument for measuring efficiency.

Example File Location

`<TestStand Public>\Examples\Interfacing with Hardware\InstrumentStudio\Efficiency\Efficiency.seq`

Highlighted Features

- IO Configuration Step Types
- Sweep Loop Step Type

Major API

None

Prerequisites

Complete the following steps before executing this example on 64-bit TestStand.

1. Install the following NI software:
 - LabVIEW (64-bit)
 - NI-DCPower
 - Measurement & Automation Explorer (MAX)
 - InstrumentStudio 2021 or later
 - MI Device Support for TestStand



Note

- MAX and InstrumentStudio are included by default when you install NI-DCPower.
- MI Device Support for TestStand is included by default with NI-DCPower only when TestStand is already installed. You can install it separately with NI Package Manager once TestStand is installed.

2. Complete the following setup:
 - In TestStand, configure the LabVIEW Adapter to use the LabVIEW Development System.
 - Create a simulated device in MAX. Right click **Devices and Interfaces»Create New »Simulated NI-DAQmx Device or Modular Instrument»Finish**.
 - In the Create Simulated NI-DAQmx Device dialog, expand the node for Power Supplies and select the NI PXIe-4143.
 - For the newly added device, update the name of the device in MAX to `dev2` and save the device.

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane of the MainSequence sequence, review the steps in the Setup step group.

- The Get Sequence File Directory step gets the directory of the example sequence file.
 - The Identify CSV Output file step creates the output stream in the sequence file directory.
 - The Create DC Power Session and Apply Configuration step creates a session (dev2) in LabVIEW.
2. On the Steps pane, review the steps in the Main step group.
 - The Sweep DC Power - Current Vout step sweeps on the voltage values for the session (dev2) on channel 0 and the current values for the session (dev2) on channel 1.
 - The Measure Current And Voltage - Channel0 step measures the current and voltage for channel 0.
 - The Measure Current And Voltage - Channel1 step measures the current and voltage for channel 1.
 - The Calculate Efficiency step calculates the efficiency based on the measured voltage and current values against the values set in the preceding two steps.
 3. On the Steps pane, review the steps in the Cleanup step group.
 - The Close Sessions step closes the sessions which have been opened in the current sequence.
 4. Select **Execute»Single Pass** to run the sequence.
 5. The voltage, current, and efficiency values are logged to <TestStand Public>\Examples\Interfacing with Hardware\InstrumentStudio\Efficiency\output.csv.

Related concepts:

- [TestStand Directory Structure](#)
- [Step Groups](#)

Measure Quiescent Current

Purpose

This example demonstrates how to use IO Configuration step types to communicate with the instrument for measuring quiescent current.

Example File Location

```
<TestStand Public>\Examples\Interfacing with Hardware\  
InstrumentStudio\Quiescent Current\QuiescentCurrent.seq
```

Highlighted Features

- IO Configuration Step Types
- Sweep Loop Step Type

Major API

None

Prerequisites

Complete the following steps before executing this example on 64-bit TestStand.

1. Install the following NI software:
 - LabVIEW (64-bit)
 - NI-DCPower
 - Measurement & Automation Explorer (MAX)
 - InstrumentStudio 2021 or later
 - MI Device Support for TestStand



Note

- MAX and InstrumentStudio are included by default when you install NI-DCPower.
- MI Device Support for TestStand is included by default with NI-DCPower only when TestStand is already installed. You can install it separately with NI Package Manager once TestStand is installed.

2. Complete the following setup:
 - In TestStand, configure the LabVIEW Adapter to use the LabVIEW Development System.
 - Create a simulated device in MAX. Right click **Devices and Interfaces»Create New »Simulated NI-DAQmx Device or Modular Instrument»Finish**.
 - In the Create Simulated NI-DAQmx Device dialog, expand the node for Power

Supplies and select the NI PXIe-4143.

- For the newly added device, update the name of the device in MAX to `dev2` and save the device.

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane of the MainSequence sequence, review the steps in the Setup step group.
 - The Get Sequence File Directory step gets the directory of the example sequence file.
 - The Identify CSV Output file step creates the output stream in the sequence file directory.
 - The Create DC Power Session and Apply Configuration step creates a session (`dev2`) in LabVIEW.
2. On the Steps pane, review the steps in the Main step group.
 - The Sweep Instrument step sweeps on the voltage values for the session (`dev2`).
 - The Measure Current And Voltage step measures the current and voltage.
3. On the Steps pane, review the steps in the Cleanup step group.
 - The Close Sessions step closes the sessions which have been opened in the current sequence.
4. Select **Execute»Single Pass** to run the sequence.
5. The voltage, current, efficiency values are logged to `<TestStand Public>\Examples\Interfacing with Hardware\InstrumentStudio\Quiescent Current\output.csv`.

Related concepts:

- [TestStand Directory Structure](#)
- [Step Groups](#)

Interfacing with NI-Hardware drivers using Python Custom step and Python Adapter

Purpose

This example demonstrates the possibility of TestStand interfacing with NI device drivers through Python packages for the desired NI devices.

The example will not use the actual device drivers. It will mimic the behaviour of the device drivers. You can modify the Python modules to interact with actual Python drivers.

The example will call into a Python module that will use simulated Python drivers for accessing device.

Example File Location

```
<TestStand Public>Examples\Interfacing with Hardware\  
Interfacing Modular Instruments Using Python Custom Step  
Type\InterfacingModularInstrumentsUsingPyCustomStepType.seq
```

Highlighted Features

- Custom Step Types

Major API

None

Prerequisites

You must have a version of Python supported by your version of TestStand installed and added to your PATH variable.

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane of the MainSequence sequence, review the steps in the Setup step group.
 - The Create Device Sessions step, which is a Sequence Call step, simulates the creation of device sessions for the NI-DMM and NI-DcPower devices.
2. On the Steps pane of the MainSequence sequence, review the steps in the Main step group.
 - The NI-DMM step, which is a Sequence Call step, simulates the interfacing with the NI-DMM Python package for the NI-DMM device.
 - The NI-SMU [DcPower] step, which is a Sequence Call step, simulates the interfacing with the NI-DcPower Python package for the NI-DCPower device.
3. On the Steps pane of the MainSequence sequence, review the steps in the Cleanup step group.
 - The Close Device Sessions step, which is a Sequence Call step, simulates the releasing of the device sessions for NI-DMM and NI-DcPower devices.
4. On the Steps pane of the CreateDeviceSessions subsequence, review the steps in the Main step group.
 - The New NI-DMM Session and the New NI-DCPower Session steps, which are instances of the Custom Step Types NIDMM_NewSession and NIDCPOWER_NewSession of the sequence file respectively, create new sessions.
 - The New NI-DMM Session step and the New NI-DCPower Session step open two separate Python interpreter sessions for the respective devices.
5. On the Steps pane of the NI-DMM subsequence, review the steps in the Main step group.
 - The Auto Schedule step, which is an Auto Schedule Synchronization step, performs auto scheduling (avoids multiple access) of the steps within the Use Auto Scheduled Resource block.
 - The Use Auto Scheduled Resource block, which is a Use Auto Scheduled Resource Synchronization step, defines the set of steps that needs to be considered for auto scheduling.
 - The Configure NI-DMM Session step, simulates the initialization of the NI-DMM device session by setting the Range, Rate, number of Points to measure and the measurement type properties in the NI-DMM device session object.
 - The Get DMM Measurements step simulates the collecting of measurement values from the NI-DMM device using the NI-DMM device session object.
 - The Average DMM Measurements, which is a Numeric Limit Test step, calculates the average of the measured values and validates it with a defined

test range.

- The Results step, which is an Additional Results step, displays the measurement values obtained from the NI-DMM session object.
6. On the Steps pane if the NI-SMU [DcPower] subsequence, review the steps in the Main step group.
- The Auto Schedule step, which is an Auto Schedule Synchronization step, performs auto scheduling (avoids multiple access) of the steps within the Use Auto Scheduled Resource block.
 - The Use Auto Scheduled Resource block, which is a Use Auto Scheduled Resource Synchronization step, defines the set of steps that needs to be considered for auto scheduling.
 - The Configure NI-DCPower Session step, simulates the initialization of the NI-DCPower device session by setting the current and voltage Limits and Ranges and the number of Points to measure in the NI-DCPower device session object.
 - The Get DCPower Measurements [Voltage, Current] step simulates the collecting of voltage and current measurement values from the NI-DCPower device using the NI-DCPower device session object.
 - The Average DCPower Voltage Measurement in Channel 0 and 1 and the Average DCPower Current Measurement in Channel 0 and 1, which is an Action step, calculates the average of the measured values across channels 0 and 1.
 - The Validate Averaged Measurements, which is a Multiple Numeric Limit Test step, validates the average of the measured values with defined test ranges.
 - the Results step, which is an Additional Results step, displays the measurement values obtained from the NI-DCPower session object across channels 0 and 1.
7. On the Steps pane of the CloseDeviceSessions subsequence, review the steps in the Main step group.
- The Close NI-DMM Session and the Close NI-DCPower Session steps, which are instances of the Custom Step Types NIDMM_CloseSession and NIDCPOWER_CloseSession respectively of the sequence file, simulate the releasing of the device sessions.
8. **Select Execute»Single Pass** to run the sequence.
9. When execution completes, review the report.

Related concepts:

- [TestStand Directory Structure](#)
- [Custom Step Types](#)

- [Step Groups](#)

Related information:

- [TestStand Python Support](#)

Session Manager

The <TestStand Public>\Examples\Interfacing with Hardware\Session Manager directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)
- [Session Manager Support for 64-bit TestStand](#)

Session Manager - LabWindows/CVI

Purpose

This example demonstrates how to use NI Session Manager to share named instrument handles between LabWindows/CVI code modules.

Example File Location

```
<TestStand Public>\Examples\Interfacing with Hardware\  
Session Manager\CVI\Session Manager.seq
```

Highlighted Features

- IVI
- NI Session Manager

Major API

None

Prerequisites

You must complete the following steps before you can run this example.

1. Ensure that you have the LabWindows/CVI Run-Time Engine installed and you configure the LabWindows/CVI Adapter to execute steps in-process. If you want to use the **Execute Steps in an External Instance of CVI** option, you must have the LabWindows/CVI development environment installed.
2. Install the following software:
 - Measurement & Automation Explorer (MAX)
 - IVI Compliance Package
3. Complete the following steps to download and install an IVI-compliant instrument driver for the IVI DMM class that the example can use, such as `hp34401a`.
 - a. Refer to the Instrument Driver Network at ni.com/idnet.
 - b. Search for the manufacturer or the type of instrument you want to download. In MAX, the driver installer automatically creates a driver session entry under the Driver Sessions item and populates the software module information under the Instrument Driver Software Module item in the Advanced folder.
4. Complete the following steps to create a logical name for the session.
 - a. In MAX, expand the **IVI Drivers** category.
 - b. Right-click the **Logical Names** item in the configuration tree and select **Create New** from the context menu.
 - c. Complete the configuration panel for the new logical name, "`SampleDMM`". Logical names are case-sensitive. In the Driver Session control, specify the driver session you downloaded.

How to Use This Example

The Get DMM Session step obtains a reference to the session named "`SampleDMM`", which the `Locals.logicalName` local variable specifies. When the step creates the session, Session Manager initializes the instrument session and the step stores the reference in the `DMM_Reference` local variable so the session must continue to exist until the sequence completes.

The remaining steps in the sequence call code modules. Each code module calls Session Manager and obtains the instrument handle for the session the `Locals.logicalName` local variable specifies. Because the Get DMM Session step

ensures that the session exists for the life of the sequence, each module uses the same session and the instrument initializes and closes only once.



Note Class drivers launch simulation dialog boxes if interactive simulation is enabled. These dialog boxes might not have taskbar items and can go behind the sequence editor. In addition, the IVI soft front panels remain visible until you close the Execution window.

Related concepts:

- [TestStand Directory Structure](#)
- [Session Manager Support for 64-bit TestStand](#)

Session Manager - LabVIEW

Purpose

This example demonstrates how to use NI Session Manager to share named instrument handles between LabVIEW code modules.

Example File Location

```
<TestStand Public>\Examples\Interfacing with Hardware\  
Session Manager\LabVIEW\Session Manager.seq
```

Highlighted Features

- IVI
- NI Session Manager

Major API

None

Prerequisites

You must complete the following steps before you can run this example.

1. Ensure that you have the LabVIEW development system installed and you configure the LabVIEW Adapter to use the LabVIEW development system.
2. Install the following software:
 - Measurement & Automation Explorer (MAX)
 - IVI Compliance Package, including the LabVIEW Support portion of the IVI Class Drivers
3. Complete the following steps to download and install an IVI-compliant instrument driver for the IVI DMM class that the example can use, such as `hp34401a`.
 - a. Refer to the Instrument Driver Network at ni.com/idnet.
 - b. Search for the manufacturer or the type of instrument you want to download. In MAX, the driver installer automatically creates a driver session entry under the Driver Sessions item and populates the software module information under the Instrument Driver Software Module item in the Advanced folder.
4. Complete the following steps to create a logical name for the session.
 - a. In MAX, expand the **IVI Drivers** category.
 - b. Right-click the **Logical Names** item in the configuration tree and select **Create New** from the context menu.
 - c. Complete the configuration panel for the new logical name, which is `"SampleDMM"`. Logical names are case-sensitive. In the Driver Session control, specify the driver session you downloaded.

How to Use This Example

The Get DMM Session step obtains a reference to the session named `"SampleDMM"`, which the `Locals.logicalName` local variable specifies. When the step creates the session, Session Manager initializes the instrument session and the step stores the reference in the `DMM_Reference` local variable so the session must continue to exist until the sequence completes.

The remaining steps in the sequence call code modules. Each code module calls Session Manager and obtains the instrument handle for the session the `Locals.logicalName` local variable specifies. Because the Get DMM Session step ensures that the session exists for the life of the sequence, each module uses the same session and the instrument initializes and closes only once.



Note Class drivers launch simulation dialog boxes if interactive simulation is

enabled. These dialog boxes might not have taskbar items and can go behind the sequence editor. In addition, the IVI soft front panels remain visible until you close the Execution window.

Related concepts:

- [TestStand Directory Structure](#)
- [Effectively Using LabVIEW with TestStand](#)
- [Session Manager Support for 64-bit TestStand](#)

Switch Executive

Description

This example demonstrates how to use TestStand to integrate and perform switching operations with NI Switch Executive.

Example File Location

```
<TestStand Public>\Examples\Interfacing with Hardware\  
Switch Executive\Switch Executive.seq
```

Highlighted Features

- NI Switch Executive integration
- IVI Switch step type
- Switching panel

Major API

None

Prerequisites

This example assumes you have installed a release or evaluation version of NI Switch Executive. This example also assumes the SwitchExecutiveExample virtual device exists and the SampleMatrix1 and SampleMatrix2 IVI logical names are configured to run in simulation mode.

The virtual device is created when you install NI Switch Executive. If the device and/or logical names have been removed, you can recover them by performing the following steps:

1. Open the Windows command line and type `cd <National Instruments>/switch executive`, where `<National Instruments>` is the NI directory, which is `C:\Program Files (x86)\National Instruments` by default.
2. Enter the command `SwitchExecutive s` to recreate the virtual device and logical names.

How to Use This Example

The sequence performs switching operations using the Switching panel on the Properties tab of the Step Settings pane and the IVI Switch step type. Using the Switching panel, any step can perform switching operations while the step executes or perform switching setup for subsequent steps.

You can monitor IVI switching operations by launching NI I/O Trace prior to running this example. You must enable tracing of calls to the IVI Switch API.

Visit ni.com/ivi for more information about IVI.

Related concepts:

- [TestStand Directory Structure](#)

Characterization of a Transistor

Purpose

This example demonstrates how you can use the Sweep Loop step to characterize a transistor.

Example File Location

```
<TestStand Public>\Examples\Interfacing with Hardware\  
TransistorCharacterizationUsingSweepLoop\
```

TransistorCharacterization.seq

Highlighted Features

- Sweep Loop Step Type

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane of the MainSequence sequence, review the steps in the Setup step group.
 - The Get Sequence File Directory step gets the directory of the example sequence file.
 - The Identify CSV Output file step creates the output stream in the sequence file directory.
 - The Write Headers to CSV Output file step adds the row that contains "Temperature (C)", "Voltage (CE)", "Current (Ic)", and "Current gain" denoting the headers for the subsequent records.
2. On the Steps pane, review the steps in the Main step group.
 - The Sweep Transistor - Temperature step sweeps on the junction temperature of the transistor.
 - The While loop containing the Measure Junction temperature step executes until the measured temperature is equal to the desired junction temperature ± 0.1 as set in the Sweep loop step above.
 - The Sweep Transistor - Voltage and Current step sweeps on various Voltage and Current parameters.
 - The Calculate Vce step calculates the Collector-Emitter voltage as the difference of voltage between Vcc and Ve.
 - The Measure Ic step simulates the measuring of Collector current.

- The Measure I_e step simulates the measuring of Emitter current.
 - The Calculate DC Current Gain (h_{FE}) step calculates the gain in current as the ratio of output current (I_c) to input current (I_e).
 - The Write Record to CSV Output file step writes the values of Temperature, V_{ce} , measured I_c and Current gain to the output CSV file for each iteration of the Sweep Loop step.
3. On the Steps pane, review the steps in the Cleanup step group.
 - The Close CSV Output file Stream reference step closes the reference to the CSV Output File Stream.
 4. Select **Execute»Single Pass** to execute the sequence.
 5. The Temperature, V_{ce} , I_c and Current gain values are logged to <TestStand Public>\Examples\Interfacing with Hardware\TransistorCharacterizationUsingSweepLoop\output.csv.

Related concepts:

- [TestStand Directory Structure](#)

Examples for Modifying Process Models

The Modifying Process Models examples demonstrate common methods for overriding or extending default process model functionality in the client sequence file. This directory includes the following example programs.

Dynamically Set the Client Sequence File

Purpose

This example demonstrates how to dynamically specify a client sequence file. This example uses the Execution.ClientFile property to set the client sequence file of the process model.

Example File Location

```
<TestStand Public>\Examples\Modifying Process Models\
Dynamically Setting the Client Sequence File\Dynamically
Set the Client Sequence File.seq
```

Highlighted Features

- TestStand API
- Process model plug-in architecture

Major API

- Engine.GetSequenceFileEx
- Execution.ClientFile
- Engine.ReleaseSequenceFileEx

Prerequisites

None

How to Use This Example

When you run this example, TestStand prompts you to select Sequence A or Sequence B to run as the client sequence file of the process model. Run the example several times using the Test UUTs or Single Pass Execution entry points and review the report, which includes information relevant to the sequence file you selected.

Open the process model plug-in file,

`DynamicClientFileExampleModelPluginAddon.seq`, associated with this example. Notice that the Model Plugin - Pre UUT sequence sets the `Execution.ClientFile` property to the file you select at run time.



Note By default, TestStand might not unload dynamically selected client files before execution completes. For more information, review the comment on the Unload Unused Client File step in the Model Plugin - Pre UUT and Model Plugin - Pre Batch sequences in `DynamicClientFileExampleModelPluginAddon.seq`.

Complete the following steps to use this example.

1. Open `<TestStand Public>\Examples\Modifying Process Models\Dynamically Setting the Client Sequence File\Example`

-Dynamically Set the Client Sequence File.seq, read the dialog box that appears, and dismiss the dialog box.

2. Select **Execute»Single Pass**. Select the sequence you want to run when prompted.
3. Review the report that TestStand displays when execution completes.
4. Repeat steps 2-3 but select **Execute»Test UUTs** to run the sequence using the Test UUTs Execution entry point.

The example copies the

DynamicClientFileExampleModelPluginAddon.seq file from the <TestStand Public>\Examples directory to the <TestStand Public>\Components\ModelsPlugins\Addons directory so that the process model automatically executes the Model Plugin - Pre UUT sequence in the file. If you want to set a breakpoint in the Model Plugin - Pre UUT sequence, set the breakpoint in the copy of the file in the <TestStand Public>\Components\ModelsPlugins\Addons directory. If you make any changes to the file in the Addons directory you must change the name of the file because the Example - Dynamically Set the Client Sequence File.seq file overwrites the DynamicClientFileExampleModelPluginAddon.seq file in the Addons directory when you load the Example - Dynamically Set the Client Sequence File.seq file and deletes the DynamicClientFileExampleModelPluginAddon.seq file from the Addons directory when you unload the Example - Dynamically Set the Client Sequence File.seq file.

Entry Point Ignores Client File Show Entry Point for All WindowsModelSequence PropertiesSample Process Model for initiating Testing Without an Active Sequence File Window.seq

Related concepts:

- [TestStand Directory Structure](#)
- [Process Model Plug-In Architecture](#)
- [Execution Entry Points in the Sequential Process Model](#)

Overriding PreUUT Model Callbacks

Purpose

This example demonstrates how to override the PreUUT Model callback sequence to simulate scanning a serial number and performing a database query. When you do not override the PreUUT callback, the default process model prompts you to manually enter a serial number.

Example File Location

```
<TestStand Public>\Examples\Modifying Process Models\  
Overriding Model Callbacks - Sequential Model\CVI\  
Overriding the PreUUT Callback.seq
```

Highlighted Features

Callbacks

Major API

None

Prerequisites

You must have the LabWindows/CVI Run-Time Engine installed and you must configure the LabWindows/CVI Adapter to execute steps in-process. If you want to use the **Execute Steps in an External Instance of CVI** option, you must have the LabWindows/CVI development environment installed.

How to Use This Example

Complete the following steps to review the sequences and steps in this example.

1. On the Sequences pane, select the `MainSequence`. The `MainSequence` contains four Numeric Limit Test steps.
2. Select **Edit»Sequence File Callbacks** to launch the Sequence File Callbacks dialog box, in which you can select a callback to override. For the PreUUT callback, the

Present column specifies yes. Click **OK** to close the Sequence File Callbacks dialog box.

3. On the Sequences pane, select the **PreUUT** callback. The IdentifyUUT step, which uses the C/C++ DLL Adapter, calls a DLL to simulate scanning a serial number and performing a database query.
4. Select **Execute»Test UUTs** to run the sequence. The PreUUT callback executes instead of the PreUUT callback in the process model, so TestStand does not launch the default UUT Information dialog box for you to manually enter a serial number.

You can click the **Stop Testing** button at any time to terminate the execution.

Related concepts:

- [Model Callbacks in the Sequential Process Model](#)
- [Process Model Architecture](#)
- [TestStand Directory Structure](#)
- [Callback Sequences](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)

Overriding PreUUT and PostUUT Parallel Model Callbacks

Purpose

This example demonstrates how to override the default UUT Information dialog box in the Parallel process model, override the ModelOptions callback to disable the default dialog box, and override the PreUUT callback to define a new method for obtaining the serial number. Optionally, you can override the PostUUT callback to obtain the status of a test socket after it finishes a run.

Example File Location

```
<TestStand Public>\Examples\Modifying Process Models\
Overriding Model Callbacks - Parallel Model\Overriding
PreUUT and PostUUT Callbacks.seq
```

Highlighted Features

Parallel process model

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to run this example.

1. Select **Execute»Test UUTs** to run the sequence. Each test socket in the parallel execution launches a UUT Information dialog box that prompts you to enter a serial number. The dialog boxes launch because the default UUT Information dialog box for the Parallel process model has been disabled.
2. Enter any serial number in each UUT Information dialog box and click **OK**.
3. Click **OK** when TestStand displays the results of each test.
4. Click **Cancel** in each UUT Information dialog box to stop executing each test socket. Each Execution window displays a report.
5. Select **Window»Close Completed Executions** to close all the Execution windows.

Complete the following steps to review the sequences and steps used in the example.

1. On the Sequences pane, select the **ModelOptions** callback sequence.
2. On the Steps pane, select the **Change setting to not show dialog** step, which is a Statement step.
3. On the Step Settings pane, click the Expression edit tab. The Expression control specifies an expression that disables the default UUT Information dialog box for the Parallel process model.
4. On the Sequences pane, select the **PreUUT** callback sequence. This sequence specifies two Action steps that use the C/C++ DLL Adapter and call a DLL that returns screen size information TestStand uses to determine where the new dialog boxes need to launch on the screen. The TestSocket Serial Num Message Popup step, which is a Message Popup step, requests the serial number for each test socket TestStand executes in parallel.

You can complete the following steps to re-enable the default UUT Information dialog

box.

1. On the Sequences pane, select the **ModelOptions** callback sequence.
2. On the Step Settings pane for the Change setting to not show dialog step, click the Properties tab.
3. Click **Run Options** on the Properties tab to show the Run Options panel.
4. Select **Skip** from the Run Mode ring control.
5. Select **Execute»Test UUTs** to run the sequence. Click No when TestStand prompts you to save changes to the sequence file. The default UUT Information dialog box launches to prompt you to enter test socket serial numbers.
6. Close the sequence file and do not save any changes.

Related concepts:

- [Model Callbacks in the Parallel Process Model](#)
- [TestStand Directory Structure](#)
- [Parallel Process Model](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)

Overriding PreUUT and PostUUT Batch Model Callbacks

Purpose

This example demonstrates how to override the PreUUT callback to customize the batch and test socket serial number entry and how to override the PostUUT callback to customize the batch and test socket status report.

Example File Location

```
<TestStand Public>\Examples\Modifying Process Models\
Overriding Model Callbacks - Batch Model\Overriding PreUUT
and PostUUT Callbacks.seq
```

Highlighted Features

Batch process model

Major API

None

Prerequisites

None

How to Use This Example

This example overrides the PreUUT and PostUUT callbacks, but the OverrideSerialNumForBatchModel1.seq example sequence file overrides the PreBatch and PostBatch callbacks.

The advantage of overriding the PreUUT and PostUUT callbacks, and not the PreBatch and PostBatch callbacks, is that TestStand executes the PreUUT and PostUUT callbacks in parallel for each test socket, and on some computers this might improve performance and allow for simpler code in the callbacks.

The disadvantages include the following:

- You can disable test sockets only in the PreBatch callback. You cannot disable test sockets in the PreUUT callback.
- Test sockets you have disabled, terminated, or aborted never call the PreUUT and PostUUT callbacks. If you want to record information for the test sockets, you must do so in the PreBatch and PostBatch callbacks.

Related concepts:

- [Model Callbacks in the Batch Process Model](#)
- [TestStand Directory Structure](#)
- [Batch Process Model](#)
- [Overriding PreBatch and PostBatch Model Callbacks](#)

Overriding PreBatch and PostBatch Model Callbacks

Purpose

This example demonstrates how to override the PreBatch callback to customize the

batch and test socket serial number entry and how to override the PostBatch callback to customize the batch and test socket status report.

Example File Location

```
<TestStand Public>\Examples\Modifying Process Models\  
Overriding Model Callbacks - Batch Model\Overriding  
PreBatch and PostBatch Callbacks.seq
```

Highlighted Features

Batch process model

Major API

None

Prerequisites

None

How to Use This Example

This example overrides the PreBatch and PostBatch callbacks, but the OverrideSerialNumForBatchModel2.seq example sequence file overrides the PreUUT and PostUUT callbacks.

The advantages of overriding the PreBatch and PostBatch callbacks, and not the PreUUT and PostUUT callbacks, include the following:

- You can enable or disable test sockets.
- You can access data for test sockets you have disabled, terminated, or aborted.

The disadvantage is that you must set or retrieve information for each test socket in a loop instead of in parallel.

Related concepts:

- [Model Callbacks in the Batch Process Model](#)

- [TestStand Directory Structure](#)
- [Batch Process Model](#)
- [Overriding PreUUT and PostUUT Batch Model Callbacks](#)

Examples for Modifying User Interfaces

The Modifying User Interfaces examples demonstrate common TestStand user interface features and customizations. This directory includes the following example programs.

Building a TestStand UI with Native Controls

The <TestStand Public>\Examples\Modifying User Interfaces\Building a TestStand UI with Native Controls directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Building a TestStand UI with Native Controls - LabWindows/CVI

Purpose

This example shows how to create a TestStand user interface using only LabWindows/CVI controls. The UI uses the TestStand UI manager controls to manage the application.

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\  
Building a TestStand UI with Native Controls\CVI\TestStand  
UI with Native Controls.prj
```

Highlighted Features

TestStand User Interface API

Major API

- `ApplicationManager.GetCommand`
- `Command.Execute`

Prerequisites

You must have LabWindows/CVI installed to execute this example.

How to Use This Example

Complete the following steps to review the code in this example:

1. In `TestStand UI with Native Controls.prj`, open **TestExec.c**.
2. Select the **Initialize** case. In this case, references to the manager controls and the LabVIEW controls are stored in the UI data cluster, and this application is set as the main TestStand window.
3. Find the `SetupActiveXControls` function. This case specifies callback functions that will execute when a TestStand event occurs. Refer to the in-line comments for more information.



Note The user interface used control callbacks to implement functionality for controls, in the same way as a typical CVI UI application. These callbacks use the `TSUI_ApplicationMgrGetCommand` and `TSUI_CommandExecute` functions to implement TestStand-specific functionality.

Complete the following steps to run the example:

1. (64-bit TestStand) Navigate to **Build»Configuration** and select **Debug64** as the configuration.
2. Click **Debug Project** to start the user interface.
3. Select **Open Sequence File** to open a sequence file, then select **Execute** to run the file using the first entry point in the current process model.
4. Once the execution completes, the report file path is added to the table control. This functionality is implemented in the `ApplicationMgr_OnEndExecution` event callback.

Related concepts:

- [TestStand Directory Structure](#)
- [TestStand API Overview](#)

Building a TestStand UI with Native Controls - LabVIEW

Purpose

This example shows how to create a TestStand user interface using only LabVIEW controls. The UI uses the TestStand UI manager controls to manage the application.

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\  
Building a TestStand UI with Native Controls\LabVIEW\  
TestStand UI with Native Controls.lvproj
```

Highlighted Features

TestStand User Interface API

Major API

- ApplicationManager.GetCommand
- Command.Execute

Prerequisites

You must have the LabVIEW Development System installed to execute this example.

How to Use This Example

Complete the following steps to review the code in this example:

1. In `TestStand UI with Native Controls.lvproj`, open **BasicUIwithNativeControls.vi**.
2. Select the **Initialize** case. In this case, references to the manager controls and the

LabVIEW controls are stored in the UI data cluster, and this application is set as the main TestStand window.

3. Select the **Register Event Callbacks** case. This case specifies callback VIs that will execute when a TestStand event occurs. Refer to the in-line comments for more information.
4. Select the **Handle Events** case. This case handles user interaction. Within the event cases, the `GetCommand` and `Command.Execute` methods are called to implement TestStand-specific functionality.
5. Select the **Shutdown** case. This case closes all references and closes the panel.

Complete the following steps to run the example:

1. Click **Run** to start the user interface.
2. Select **Open Sequence File** to open a sequence file, then select **Execute** to run it using the first entry point in the current process model.
3. Once the execution completes, the report file path is added to the table control. This functionality is implemented in the `EndExecution` event callback.

Related concepts:

- [TestStand Directory Structure](#)
- [TestStand API Overview](#)

Building a TestStand UI with Native Controls - .NET

Purpose

This example shows how to create a TestStand user interface using only .NET controls. The UI uses the TestStand UI manager controls to manage the application.

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\
Building a TestStand UI with Native Controls\dotNET\
TestStand UI with Native Controls.sln
```


Highlighted Features

TestStand User Interface API

Major API

- `ApplicationManager.GetCommand`
- `Command.Execute`

Prerequisites

You must have Visual Studio installed to execute this example.

How to Use This Example

Complete the following steps to review the code in this example:

1. In `TestStand UI with Native Controls.sln`, right-click **MainForm.cs**.
2. Select the **ApplicationManager** control, then click **Events** in the properties pane. Notice that callbacks have been specified for certain TestStand events. Double-click a callback to view the code.



Note The user interface used control callbacks to implement functionality for controls, in the same way as a typical .NET UI application. These callbacks use the `axApplicationMgr.GetCommand` and `Command.Execute` methods to implement TestStand-specific functionality.

Complete the following steps to run the example:

1. Click **Start Debugging** to start the user interface.
2. Select **Open Sequence File** to open a sequence file, then select **Execute** to run the file using the first entry point in the current process model.
3. Once the execution completes, the report file path is added to the table control. This functionality is implemented in the `axApplicationMgr_EndExecution` event callback.

Related concepts:

- [TestStand Directory Structure](#)
- [TestStand API Overview](#)

Connecting UI Controls Demo

Purpose

The Connecting Controls User Interface demonstrates the available TestStand User Interface controls and allows you to customize their functionality. The TestStand User Interface controls get their functionality through a connection to a manager control. The manager controls provide methods to implement the various types of connections.

This example allows you to explore the available connections and see their behavior in real time.

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\  
Connecting UI Controls\DotNet\bin\Release\TestExec.exe
```

Highlighted Features

TestStand User Interface Controls

Major API

UI Connection methods

Prerequisites

None

How to Use This Example

To launch the User Interface, Open the **TestExec.exe** executable. The UI automatically opens a sequence file and runs a sequence, then pauses the sequence at a breakpoint.

Navigate the tabs of the interface to view the different UI controls available with TestStand and the types of connections to manager controls that they support. Each tab provides more information about each of the following UI Control connection types:

- **View Connections**—Activates a SequenceView, VariablesView, or ReportView control. The manager controls provide a separate method for each connection type, so no `cmdKind` parameter is needed.
- **List Connections**—Configures a TestStand ComboBox, ListBarPage, or ListBox to display a TestStand List, such as the sequences in the current file. The manager controls provide a separate method for each connection type, so no `cmdKind` parameter is needed.
- **Command Connections**—Configures a TestStand Button or Checkbox control to execute a specified command when clicked. Change the enumerations to see the behavior of the available command kinds.
- **Information Source Connections**—Displays caption, image, and numeric information on Label controls, ExpressionEdit controls, and StatusBar panes. Changing the source Input will change what information is displayed in the connected control.

Click the **Additional Information** button in the example UI to view more information about these connection types.

Related concepts:

- [TestStand Directory Structure](#)
- [TestStand API Overview](#)
- [Connecting Manager Controls to Visible Controls](#)

Creating a Basic User Interface

The <TestStand Public>\Examples\Modifying User Interfaces\Creating a Basic User Interface directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Creating a Basic User Interface - LabWindows/CVI

Purpose

This example provides a simplified user interface to demonstrate the architecture of a TestStand user interface. The interface code demonstrates the following actions in a TestStand interface:

- Provide functionality to TestStand controls by connecting them to manager controls
- Register callbacks to execute when TestStand events occur

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\  
Creating a Basic User Interface\CVI\Basic User  
Interface.prj
```

Highlighted Features

- TestStand User Interface Controls
- TestStand User Interface API

Major API

TestStand User Interface API

Prerequisites

You must have LabWindows/CVI installed to execute this example.

How to Use This Example

Complete the following steps to run the example:

1. (64-bit TestStand) Navigate to **Build»Configuration** and select **Debug64** as the configuration.
2. Click **Debug Project** to start the user interface.

3. Notice that the control functionality matches the specified connections in the SetupActiveXControls function.
4. Use the interface to open a sequence file and execute it. The execution displays in the executionView control if tracing is enabled.

Complete the following steps to review the sequences and steps in this example:

1. In Basic User Interface.prj, open **TestExec.c**.
2. Find the SetupActiveXControls function. This function registers callback functions that will execute when a TestStand event occurs. Also, the TestStand UI controls are connected to manager controls to give them functionality. Refer to the in-line comments for more information.

Related concepts:

- [TestStand Directory Structure](#)
- [TestStand API Overview](#)

Creating a Basic User Interface - LabVIEW

Purpose

This example provides a simplified user interface to demonstrate the architecture of a TestStand user interface. The interface code demonstrates the following actions in a TestStand interface:

- Initialize the TestStand engine
- Provide functionality to TestStand controls by connecting them to manager controls
- Register callbacks to execute when TestStand events occur
- Shut down TestStand

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\  
Creating a Basic User Interface\LabVIEW\Basic User  
Interface.lvproj
```

Highlighted Features

- TestStand User Interface Controls
- TestStand User Interface API

Major API

TestStand User Interface API

Prerequisites

You must have the LabVIEW Development System involved to execute this example.

How to Use This Example

Complete the following steps to run the example:

1. Click **Run** to start the user interface.
2. Notice that the control functionality matches the specified connections in the Configure Connections case.
3. Use the interface to open a sequence file and execute it. The execution displays in the executionView control if tracing is enabled.

Complete the following steps to review the sequences and steps in this example:

1. In Basic User Interface.lvproj, open **TestStand Basic User interface.vi**.
2. Select the **Initialize** case. In this case, references to the UI controls are stored in the UI data cluster, and this application is set as the main TestStand window.
3. Select the **Configure Connections** case. In this case, the TestStand UI controls are connected to manager controls to give them functionality. Refer to the in-line comments for more information.
4. Select the **Register Event Callbacks** case. This case specifies callback VIs that will execute when a TestStand event occurs. Refer to the in-line comments for more information.
5. Select the **Handle Events** case. This case handles LabVIEW specific events, such as closing the panel. Refer to the in-line comments for more information.
6. Select the **Shutdown** case. This case closes all references and closes the panel.

Related concepts:

- [TestStand Directory Structure](#)
- [TestStand API Overview](#)

Creating a Basic User Interface - .NET

Purpose

This example provides a simplified user interface to demonstrate the architecture of a TestStand user interface. The interface code demonstrates the following actions in a TestStand interface:

- Provide functionality to TestStand controls by connecting them to manager controls
- Register callbacks to execute when TestStand events occur

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\  
Creating a Basic User Interface\DotNet\Basic User  
Interface.sln
```

Highlighted Features

- TestStand User Interface Controls
- TestStand User Interface API

Major API

TestStand User Interface API

Prerequisites

You must have Visual Studio installed to execute this example.

How to Use This Example

Complete the following steps to run the example:

1. Click **Run** to start the user interface.
2. Notice that the control functionality matches the specified connections in the `MainForm_Load` method.
3. Use the interface to open a sequence file and execute it. The execution displays in the `executionView` control if tracing is enabled.

Complete the following steps to review the sequences and steps in this example.

1. Open **Basic User Interface.sln** in Visual Studio, then right click **MainForm.cs** and select **View Code**.
2. Find the `MainForm_Load` method. Within this method, the TestStand UI controls are connected to manager controls to give them functionality. Refer to the in-line comments for more information.
3. Double click **MainForm.cs** to view the designer. Select the `ApplicationManager` control, then click **Events** in the properties pane. Notice that callback functions have been specified for certain TestStand events. Double click the event callback to view the callback code.

Related concepts:

- [TestStand Directory Structure](#)
- [TestStand API Overview](#)

Handling UI Messages

The `<TestStand Public>\Examples\Modifying User Interfaces\Handling UI Messages` directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Handling UI Messages - LabWindows/CVI

Purpose

This example shows how you can use custom UI messages to communicate between the client sequence file and the user interface.

- **Sending data from the sequence to the UI**—The Sequence posts UI messages containing a status string. The UI message handler reads the string and updates a UI control with the data.
- **Sending data from the UI to the sequence**—Before executing, the sequence posts a message requesting the state of the "show dialogs" UI setting. The UI message handler sets the Locals.ShowDialogs sequence property using the TestStand API, based on the value specified in the UI.



Note In order for the UI messages posted in this sequence to be handled, you must execute the sequence in the "Basic User Interface with UI Message Handler" Interface, located in the language specific subfolders of this sequence file. While the sequence will run successfully in the Sequence Editor or other user interfaces, the user messages posted will have no effect.

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\  
Handling UI Messages\CVI\Using Custom UI Messages.prj
```

Highlighted Features

- User Interface Messages
- TestStand User Interface controls

Major API

Thread.PostUIMessageEx

Prerequisites

You must have LabWindows/CVI installed to execute this example.

How to Use This Example

Complete the following steps to run the example:

1. (64-bit TestStand) Navigate to **Build»Configuration** and select **Debug64** as the configuration.
2. Click **Debug Project** in CVI to start the user interface. The UI Message example sequence is opened by the UI automatically.
3. Run the sequence. Notice that the sequence status is updated as each UI message is posted with ID 10100.
4. After the execution completes, uncheck the **Show Sequence Dialogs** option on the user interface. Run the sequence again, and notice that the message dialogs no longer appear. The dialogs no longer appear because the Sequence File posts UI message 10200. The handler for this message sets the FileGlobals.ShowDialogs property to the value of the Show Sequence Dialogs setting in the setup step group.
5. You can optionally execute this sequence file in other user interfaces that do not handle the specified user messages. The sequence will run successfully, but the user messages posted will have no effect.

Complete the following steps to review the UI message handling code in the CVI UI:

1. In `Using Custom UI Messages.prj`, open `TestExec.c`.
2. Find the function `TSUI__ApplicationMgrEventsRegOnUserMessage`. This function registers the `ApplicationMgr_OnUserMessage` function as the User Message message handler. The User Message Event begins when a UI message with ID greater than 10000 is posted. These IDs are reserved for user-generated messages.
3. Find the `ApplicationMgr_OnUserMessage` function. Notice that each UI message is defined as a case in the switch block. These cases define how each UI message is handled.

Complete the following steps to review how UI messages are posted from an executing sequence:

1. Open the example sequence file <TestStand Public>\Examples\Modifying User Interfaces\Handling UI Messages\UI Message Example.seq
2. Inspect the statement steps. Each of these steps posts a UI message using the `Thread.postMessageEx` method.

Related concepts:

- [TestStand Directory Structure](#)
- [User Interface Messages \(UIMessages\)](#)
- [TestStand API Overview](#)

Handling UI Messages - LabVIEW

Purpose

This example shows how you can use custom UI messages to communicate between the client sequence file and the user interface.

- **Sending data from the sequence to the UI**—The Sequence posts UI messages containing a status string. The UI message handler reads the string and updates a UI control with the data.
- **Sending data from the UI to the sequence**—Before executing, the sequence posts a message requesting the state of the "show dialogs" UI setting. The UI message handler sets the `Locals.ShowDialogs` sequence property using the TestStand API, based on the value specified in the UI.



Note In order for the UI messages posted in this sequence to be handled, you must execute the sequence in the "Basic User Interface with UI Message Handler" Interface, located in the language specific subfolders of this sequence file. While the sequence will run successfully in the Sequence Editor or other user interfaces, the user messages posted will have no effect.

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\
Handling UI Messages\LabVIEW\Using Custom UI
```

Messages.lvproj

Highlighted Features

- User Interface Messages
- TestStand User Interface controls

Major API

Thread.PostUIMessageEx

Prerequisites

You must have the LabVIEW Development System installed to view this example.

How to Use This Example

Complete the following steps to run the example:

1. In `Using Custom UI Messages.lvproj`, open `Basic User Interface with UI Message Handler.vi`.
2. Click the **Run** button on the VI to start the user interface. The UI Message example sequence is opened by the UI automatically.
3. Run the sequence. Notice that the sequence status is updated as each UI message is posted with ID 10100.
4. After the execution completes, uncheck the **Show Sequence Dialogs** option on the user interface. Run the sequence again and notice that the message dialogs no longer appear. The dialogs no longer appear because the Sequence File posts UI message 10200. The handler for this message sets the `FileGlobals.ShowDialogs` property to the value of the `Show Sequence Dialogs` setting in the setup step group.
5. You can optionally execute this sequence file in other user interfaces that do not handle the specified user messages. The sequence will run successfully, but the user messages posted will have no effect.

Complete the following steps to review the UI message handling code in the UI:

1. In `Using Custom UI Messages.lvproj`, open `Basic User`

Interface with UI Message Handler.vi.

2. In the Register Event Callbacks case, notice that `UserMessage Event Callback.vi` is registered to the `UserMessage` event.
3. Open the `UserMessage Event Callback.vi` and review the message handling code in each case.

Complete the following steps to review how UI messages are posted from an executing sequence:

1. Open the example sequence file `<TestStand Public>\Examples\Modifying User Interfaces\Handling UI Messages\UI Message Example.seq`
2. Inspect the statement steps. Each of these steps posts a UI message using the `Thread.postMessageEx` method.

Related concepts:

- [TestStand Directory Structure](#)
- [User Interface Messages \(UIMessages\)](#)
- [TestStand API Overview](#)

Handling UI Messages - .NET

Purpose

This example shows how you can use custom UI messages to communicate between the client sequence file and the user interface.

- **Sending data from the sequence to the UI**—The Sequence posts UI messages containing a status string. The UI message handler reads the string and updates a UI control with the data.
- **Sending data from the UI to the sequence**—Before executing, the sequence posts a message requesting the state of the "show dialogs" UI setting. The UI message handler sets the `Locals.ShowDialogs` sequence property using the TestStand API, based on the value specified in the UI.



Note In order for the UI messages posted in this sequence to be handled,

you must execute the sequence in the "Basic User Interface with UI Message Handler" Interface, located in the language specific subfolders of this sequence file. While the sequence will run successfully in the Sequence Editor or other user interfaces, the user messages posted will have no effect.

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\  
Handling UI Messages\DotNet\Handling UI Messages.sln
```

Highlighted Features

- User Interface Messages
- TestStand User Interface controls

Major API

Thread.PostUIMessageEx

Prerequisites

You must have Visual Studio installed to execute this example.

How to Use This Example

Complete the following steps to run the example:

1. Click **Start Debugging** in Visual Studio to start the user interface. The UI Message example sequence is opened by the UI automatically.
2. Run the sequence. Notice that the sequence status is updated as each UI message is posted with ID 10100.
3. After the execution completes, uncheck the **Show Sequence Dialogs** option on the user interface. Run the sequence again and notice that the message dialogs no longer appear. The dialogs no longer appear because the Sequence File posts UI message 10200. The handler for this message sets the FileGlobals.ShowDialogs property to the value of the Show Sequence Dialogs setting in the setup step group.

4. You can optionally execute this sequence file in other user interfaces that do not handle the specified user messages. The sequence will run successfully, but the user messages posted will have no effect.

Complete the following steps to review the UI message handling code in the UI:

1. In `Handling UI Messages.sln`, open `MainForm.cs`.
2. In the form designer, select the Application manager control. In the properties pane, click the **Events** icon to view the registered events for the `ApplicationManager`.
3. Find the User message event, and notice that the `axApplicationMgr_UserMessageCallback` method is registered to handle this event. The UserMessage Event begins when a UI message with ID greater than 10000 is posted. These IDs are reserved for user-generated messages.
4. Double click the **ApplicationMgr_OnUserMessage** to navigate to the method code. Notice that each UI message is defined as a case in the switch block. These cases define how each UI message is handled.

Complete the following steps to review how UI messages are posted from an executing sequence:

1. Open the example sequence file `<TestStand Public>\Examples\Modifying User Interfaces\Handling UI Messages\UI Message Example.seq`
2. Inspect the statement steps. Each of these steps posts a UI message using the `Thread.postMessageEx` method.

Related concepts:

- [TestStand Directory Structure](#)
- [User Interface Messages \(UIMessages\)](#)
- [TestStand API Overview](#)

Updating the Status Bar Using UI Messages

The `<TestStand Public>\Examples\Modifying User Interfaces\Updating the Status Bar Using UI Messages` directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Updating the Status Bar Using UI Messages - LabWindows/CVI

Purpose

This example demonstrates how to use LabWindows/CVI to show step execution progress and status in the TestStand Sequence Editor or Execution window.

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\  
Updating the Status Bar Using UI Messages\CVI\Updating the  
Status Bar Using UI Messages.seq
```

Highlighted Features

- LabWindows/CVI Adapter
- Execution

Major API

- SequenceContext
- Thread
- Thread.PostUIMessage
- UIMessageCodes Enumeration

Prerequisites

None

How to Use This Example

Complete the following steps to use this example.

1. Open the `DisplayingProgressAndStatus.c` source file, located in the

<TestStand Public>\Examples\Modifying User Interfaces\
Updating the Status Bar Using UI Messages\CVI directory. The
source file specifies the following functionality:

- **Lines 29 and 46**—The `tsErrChkMsgPopup` function uses `TS_ThreadPostUIMessage` to post a `ProgressPercent` user interface (UI) message that updates the progress bar the sequence displays in TestStand with the percent complete.
 - **Lines 33 and 43**—The `tsErrChkMsgPopup` function uses `TS_ThreadPostUIMessage` to post a `ProgressText` UI message to update the progress text on the status bar with the percent finished.
2. Close the source file. Do not save any changes.
 3. In TestStand, select **Execute»Single Pass** to run the sequence. Review the progress bar and progress status field on the status bar of the sequence editor.

Related concepts:

- [TestStand Directory Structure](#)
- [Programming with the TestStand API in LabWindows/CVI](#)
- [Executions](#)

Updating the Status Bar Using UI Messages - LabVIEW

Purpose

This example demonstrates how to use LabVIEW to show step execution progress and status in the TestStand Sequence Editor or Execution window.

Example File Location

<TestStand Public>\Examples\Modifying User Interfaces\
Updating the Status Bar Using UI Messages\LabVIEW\Updating
the Status Bar Using UI Messages.seq

Highlighted Features

- LabVIEW Adapter
- Execution

Major API

- `SequenceContext`
- `Thread`
- `Thread.PostUIMessage`
- `UIMessageCodes` Enumeration

Prerequisites

You must have the LabVIEW development system installed and you must configure the LabVIEW Adapter to use the LabVIEW development system.

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane, select the **Call Long Test** step, which is an Action step that uses the LabVIEW Adapter.
2. On the Step Settings pane, click the LabVIEW Module tab.
3. Click the **Edit VI** button, located to the right of the VI Path control, to open in LabVIEW the VI the Call Long Test step calls.
4. In LabVIEW, switch to the block diagram and select **Help»Show Context Help** to display the Context Help window for the block diagram. The block diagram specifies the following functionality for the VI:
 - The Sequence Context property node obtains a reference to the `Thread` object and initializes the termination monitor.
 - The execution enters the Stacked Sequence Structure and executes frame 0. Select frame 0 of the Stacked Sequence Structure. The Start Time is stored in the sequence local variable.
 - Select frame 1 of the Stacked Sequence Structure. The Get Monitor Termination Status VI stops the VI if you terminate or abort the execution. If you do not terminate or abort the execution, the False case of the case structure executes.
 - The VI invokes the `Thread.PostUIMessage` method and posts a `ProgressPercent` user interface (UI) message that updates the progress bar the sequence displays in TestStand with the percent complete.
 - The VI posts a `ProgressText` UI message to update the progress text on the

status bar with the percent finished. This behavior continues in the While loop until the time limit expires or you terminate or abort the execution. If the time limit expires, the execution continues to frame 2 of the Stacked Sequence Structure.

- Select frame 2. In the False case of the case structure, the progress bar updates to 100 percent and the progress text updates to Test Finished.
- The `Thread` object reference the terminator monitor close.

5. Close the VI. Do not save any changes.

6. In TestStand, select **Execute»Single Pass** to run the sequence. Review the progress bar and progress status field on the status bar of the sequence editor.

Related concepts:

- [TestStand Directory Structure](#)
- [Programming with the TestStand API in LabVIEW](#)
- [Executions](#)

Updating the Status Bar Using UI Messages - MFC

Purpose

This example demonstrates how to use C++ to show step execution progress and status in the TestStand Sequence Editor or Execution window.

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\
Updating the Status Bar Using UI Messages\MFC\Updating the
Status Bar Using UI Messages.seq
```

Highlighted Features

- C/C++ DLL Adapter
- Execution

Major API

None

Prerequisites

None

How to Use This Example

AppWizard creates `DisplayingProgressAndStatus.dll`, which demonstrates the basics of using the Microsoft Foundation Class (MFC) Library and serves as a starting point for writing a custom DLL.

The following files, located in the `<TestStand Public>\Examples\Modifying User Interfaces\Updating the Status Bar Using UI Messages\MFC` directory, make up `DisplayingProgressAndStatus.dll`:

- `DisplayingProgressAndStatus.h`—The main header file that declares the `CDisplayingProgressAndStatusApp` class for the DLL.
- `DisplayingProgressAndStatus.cpp`—The main DLL source file that contains the `CDisplayingProgressAndStatusApp` class.
- `DisplayingProgressAndStatus.rc`—A list of all the Microsoft Windows resources the program uses, including icons, bitmaps, and cursors stored in the `res` subdirectory. You can directly edit this file in Microsoft Developer Studio.
- `res\DisplayingProgressAndStatus.rc2`—Contains resources Developer Studio does not edit. Place all resources the resource editor cannot edit in this file.
- `DisplayingProgressAndStatus.def`—Contains DLL information that must be provided to run with Windows. This file defines parameters such as the name and description of the DLL and exports functions from the DLL.
- `DisplayingProgressAndStatus.clw`—Contains information ClassWizard uses to edit existing classes, add new classes, and store information required to create and edit message maps and dialog data maps and to create prototype member functions.

Other standard files located in the `<TestStand Public>\Examples\Modifying User Interfaces\Updating the Status Bar Using UI Messages\MFC` directory include the following:

- `StdAfx.h` and `StdAfx.cpp`—Used to build a precompiled header (PCH) file named `DisplayingProgressAndStatus.pch` and a precompiled types file

named `StdAfx.obj`.

- `Resource.h`—The standard header file that defines new resource IDs. Developer Studio reads and updates this file.



Note AppWizard uses `TODO :` to indicate parts of the source code to which you must add or that you must customize.

Related concepts:

- [TestStand Directory Structure](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)
- [Executions](#)

Updating the Status Bar Using UI Messages - .NET

Purpose

This example demonstrates how to show step execution progress and status in the TestStand Sequence Editor or Execution window.

Example File Location

```
<TestStand Public>\Examples\Modifying User Interfaces\
Updating the Status Bar Using UI Messages\DotNET\Updating
the Status Bar Using UI Messages.seq
```

Highlighted Features

- Execution
- .NET Adapter

Major API

- `SequenceContext`
- `Thread`
- `Thread.PostUIMessage`
- `UIMessageCodes` Enumeration

Prerequisites

None

How to Use This Example

In TestStand, select **Execute»Single Pass** to run the sequence. Review the progress bar and progress status field on the status bar of the sequence editor.

Related concepts:

- [TestStand Directory Structure](#)
- [Executions](#)

Examples for Parallel Testing

The Parallel Testing examples demonstrate TestStand features for running your test code in parallel. This directory includes the following example programs.

Comparing Resource Usage Strategies

Description

This example demonstrates how to use the Execution Profiler to measure and understand how different multithreaded resource usage strategies affect the performance of a simultaneous test of four UUTs when each UUT requires a DMM measurement and a scope measurement.

Example File Location

```
<TestStand Public>\Examples\Parallel Testing\Comparing  
Resource Usage Strategies\Comparing Resource Usage  
Strategies.seq
```

Highlighted Features

- Execution Profiler
- Synchronization step types

Major API

N/A

Prerequisites

None

How to Use This Example

When you open the `Comparing Resource Usage Strategies.seq` example sequence file, TestStand also launches the Execution Profiler window.

Complete the following steps to walk through the example:

1. In the TestStand Sequence Editor, select the **MainSequence** and select **Run MainSequence** from the Execute menu. In the Execution Profiler window, view the instrument usage in the Item Usage per Thread graph as the example executes five sequences with five different strategies for using the DMM and scope resources. Make the profiler window wider if your screen has room. The first row of the graph shows per step locks on the top level sequence call steps. The example has these locks to better highlight the duration of each strategy in the profiler. Below the square for each lock, you can see the usage of the DMM and scope resources for that strategy.
2. For Sequential, note that each UUT uses the DMM in turn, and then each UUT uses the scope in turn.
3. For Parallel, note that each UUT uses the DMM as soon as it is available and then uses the scope as soon as it is available.
4. For Auto-scheduled, note that each UUT first uses an instrument of either kind, as soon as one is available, and then uses the other kind of instrument as soon as it is available.
5. For Parallel With Additional DMM, note that each UUT first uses either one of two DMMs as soon as one is available and then uses the scope as soon as it is available.
6. For Auto Scheduled With Additional DMM, note that each UUT first uses any instrument, as soon as one is available. It then uses the remaining type of instrument as soon as one is available.
7. Note that each strategy completes in a shorter time than the previous strategy.
8. Open the **View** menu in the profiler. To focus on the instrument usage only, the

example unchecked the following menu items when it launched the profiler in its `SequenceFileLoad` sequence:

- Show Blocked Threads
- Show Code Modules»Step
- Show Code Modules»Step Type
- Show Code Modules»Load
- Show Code Modules»Unload
- Show Synchronization»Show Auto Scheduling
- Show Synchronization»Show Batch Synchronization
- Show Synchronization»Show Wait Operations
- Show Steps
- Show UUTs, Batches, and Lots

Toggle each menu item on and off to observe the additional data the profiler displays when each item is on.

Complete the following steps to walk through the example:

1. Checkmark the following items in the View menu:
 - Show Blocked Threads
 - Show Synchronization»Show Auto Scheduling
 - Show Synchronization»Show Batch Synchronization
 - Show Synchronization»Show Wait Operations
 - Show UUTs, Batches, and Lots
2. Click **Clear** and complete the following steps:
 - a. In the TestStand Sequence Editor, right-click the Sequential step in the Main step group of the MainSequence and select **Run Selected Steps** from the context menu.
 - b. In the Execution Profiler window, checkmark **Auto Fit** below the Item Usage Per Thread graph. The Sequential strategy example sequentially runs the DMM test and then the scope test on each UUT. For each test socket thread, the Item Usage per Thread graph shows the use of the DMM resource and the scope resource. The red rectangles in the graph whose tool tips contain Batch Synchronization indicate threads waiting to enter and exit Batch Serial Sections, which is the mechanism this example uses to enforce sequential use of the DMM and scope resources. You can hide the Batch Synchronization operations by unchecking **View»Show Batch Synchronized Sections**. The Sequential strategy example uses Wait steps to simulate instrument usage. In

addition, the Sequential step in the MainSequence is configured to wait for the execution it launches to complete. You can hide the Wait operations by unchecking **View»Show Wait Operations**. The graph also shows that none of the resources are used in parallel. The time displayed when you hover over the square labeled <No Serial Number> (Batch) is 30 seconds, which is the time it took to test the batch of UUTs. The proceeding steps attempt to lower the total time span value.

- c. Click the **Clear** button to remove the graph information.
- d. In the sequence editor, right-click the Parallel step in the Main step group of the MainSequence and select **Run Selected Steps** from the context menu. In the Execution Profiler window, notice that all threads attempt to use the DMM, but only one thread acquires the DMM while the other threads block. As each thread completes the DMM test, another thread can acquire the DMM. You can hide the time that threads block while waiting to acquire resources from the graph in the profiler window display by unchecking **View»Show Blocked Threads**. Because the DMM and scope resources are used in parallel much of the time, the length of the Batch reduces to 21 seconds. However, because each thread first uses the DMM and then uses the scope, no parallelism exists until the first thread releases the DMM.
- e. Click the **Clear** button to remove the graph information and select **View»Show Blocked Threads** to show the time that threads block while waiting to acquire resources.
- f. In the sequence editor, right-click the Auto Schedule step in the Main step group of the MainSequence and select **Run Selected Steps** from the context menu. In the profiler, notice that the graph now includes rectangles labeled `Auto Schedule ("Resource name")`, which correspond to Use Auto Scheduled Resource steps that define resource usage sections in the sequence. The graph shows that the Use Auto Scheduled Resource steps block the execution of their threads until the steps can lock the resources that their section of the sequence requires. You can hide the operations of the Auto Schedule and Use Auto Scheduled Resource steps by unchecking **View»Show Auto Scheduling**. Notice that for some threads, the Auto Schedule example reorders the use of the DMM and scope resources to reduce the batch time to approximately 18 seconds, because each DMM test takes a minimum of 4.5 seconds. In this example, the test time is now resource limited, and any further improvements will require additional resources.
- g. Click the **Items** tab to display the Items table.

- h. Uncheck **View»Show UUTs, Batches, and Lots**.
- i. Click the **% Actual Time in Use** column heading twice to sort the percent values in descending order, as indicated by an inverted triangle icon. The Auto Scheduled item has the highest % Actual Time in Use value. However, this is only the name of the lock the example holds to force the profiler to display the name of the strategy being used. Ignoring that item, the DMM item now has the highest Percent Time in Use value, which indicates that system performance might improve if you add additional DMM resources.
- j. Click the **Clear** button to remove the graph information and uncheck **View»Display Blocked Threads**. Checkmark **View»Show UUTs, Batches, and Lots**.
- k. In the sequence editor, right-click the Parallel with Additional DMM step in the Main step group of the MainSequence and select **Run Selected Steps** from the context menu. In the profiler, notice that the additional DMM reduces the value of the batch time to approximately 17 seconds. Because the DMM test always precedes the scope test, the scope test cannot execute in parallel until a thread completes its DMM test.
- l. Click the **Clear** button to remove the graph information.
- m. In the sequence editor, right-click the Auto Scheduled with Additional DMM step in the Main step group of the MainSequence and select **Run Select Steps** from the context menu. In the profiler, notice that the batch time reduces to approximately 12 seconds, which is a substantial improvement over the initial value of 30 seconds for the Sequential strategy example, in which no parallel resource usage existed.
- n. In the sequence editor, open and examine the Sequential, Parallel, Auto Scheduled, Parallel with Additional DMM, and Auto Scheduled with Additional DMM sequences to view how each sequence synchronizes access to the DMM and scope resources. Notice that the DMM and scope are represented by Wait steps. The example models tests that are resource-bound instead of CPU-bound, and is an accurate model for tests that must wait on instrument and I/O operations and for multi-core systems in which the number of cores equals or exceeds the number of UUTs. You can change the values of the `DMMSimulatedTestDuration` and `ScopeSimulatedTestDuration` file global variables to determine the effect of different resource speeds on total throughput. The example contains a `ModelOptions` callback sequence that sets the number of UUTs to four. You can edit the callback sequence, or you can delete the sequence and select **Configure»Model Options** to launch the Model

Options dialog box and modify the value of the **Number of Test Sockets** option to experiment with the effect of changing the number of UUTs.

Related concepts:

- [TestStand Directory Structure](#)

Executing Code Modules in Parallel

Purpose

This example demonstrates how to run code modules in parallel. This sequence uses two Sequence Call steps that each call a subsequence in a new execution. Each subsequence calls a LabWindows/CVI code module to display a panel, which is updated by a loop in the `MainSequence` asynchronous from the original execution.

Example File Location

```
<TestStand Public>\Examples\Parallel Testing\Executing Code  
Modules In Parallel\Executing Code Modules In Parallel.seq
```

Highlighted Features

- Sequence Call steps
- Wait steps

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane, select the **Launch Display Panel 1** step, which is a Sequence

Call step.

2. On the Step Settings pane, click the Sequence Call Module tab. The Execution Options ring control specifies the **Use New Execution** option.
3. Click the **Sequence Call Advanced Settings** button, located to the right of the Execution Options ring control, to launch the Sequence Call Advanced Settings window, in which the options change depending on the execution option you select. The Wait for Execution to Complete ring control specifies the **Do not wait** option. This sequence uses the **Do not wait** option and a set of Wait steps at the end of the `MainSequence` to collect results from each subsequence.
4. On the Steps pane, select the **Wait on Panel 1 for Results** step, which is a Wait step.
5. On the Step Settings pane, click the Wait Settings edit tab. The Specify by Sequence Call option specifies that the step waits for the Launch Display Panel 1 sequence call to finish executing. Similarly, the Wait on Panel 2 for Results step waits for the Launch Display Panel 2 sequence call to finish executing.
6. On the Sequences pane, select the **Display 1** sequence.
7. On the Steps pane, select the **Display Panel 1** step, which is an Action step that uses the LabWindows/CVI Adapter.
8. On the Step Settings pane, click the LabWindows/CVI Module tab. This step calls a DLL that polls the `Counter1` file global variable and displays the content. Similarly, the Display Panel 2 step in the Display 2 sequence polls the `Counter2` file global variable and displays the content.
9. On the Sequences pane, select the `MainSequence` and select **Execute»Single Pass** to run the sequence. TestStand begins three executions: one execution updates the `Counter1` and `Counter2` variables, and the other executions poll the values of the variables.
10. Click **Stop** for both Display Panel 1 and Display Panel 2. Scroll to the bottom of the report, which displays results for the Display Panel 1 and Display Panel 2 steps under the Wait steps for each new execution. If the sequence did not use Wait steps, TestStand would not log results for the steps run in new executions.

Related concepts:

- [TestStand Directory Structure](#)
- [Programming with the TestStand API in LabWindows/CVI](#)

Executing Sequences in Parallel

Purpose

This example demonstrates how to launch a parallel, independent sequence execution with a Sequence Call step in conjunction with a Wait step so that the execution results of the primary sequence are combined with the execution results of the subsequence.

Alternatively, you can run the subsequence in the same execution as the primary sequence. Review the report TestStand generates after either scenario to see the difference in where the results are collected for the Sequence Call steps.

Example File Location

```
<TestStand Public>\Examples\Parallel Testing\Executing  
Sequences In Parallel\Executing Sequences In Parallel.seq
```

Highlighted Features

- Sequence Call steps
- Wait steps

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to use this example.

1. On the Steps pane, select the **Parallel Call** step, which is a Sequence Call step.
2. On the Step Settings pane, click the Sequence Call Module tab. The Execution Options ring control specifies the **Use New Execution** option.
3. On the Steps pane, select the **Sequential Call** step, which is a Sequence Call step.
4. On the Step Settings pane, click the **Sequence Call Module** tab. The Execution

Options ring control specifies the **None** option.

5. On the Steps pane, select the **Wait** step.
6. On the Step Settings pane, click the Wait Settings edit tab. The Specify by Sequence Call option specifies that the step waits for the Parallel Call sequence call to finish executing.
7. Select **Execute»Single Pass** to run the sequence.
8. Review the information in the dialog box that launches, and click the **Run DemoSequence in New Execution** button. A dialog box launches for each execution simultaneously. You may have to move or click **OK** in each dialog box to see the others.
9. Click **OK** in each dialog box.
10. Review the report. The results for the Message Popup step from the DemoSequence appears after the Wait step.
11. Select **Execute»Restart** to restart the `MainSequence` execution.
12. Review the information in the dialog box that launches, and click the **Run DemoSequence Sequentially** button. Now, only one dialog box launches at a time.
13. Click **OK** in each dialog box, and review the report. The Message Popup step from the DemoSequence now appears after the Message Popup 2 step.

Related concepts:

- [TestStand Directory Structure](#)

Testing UUTs in Parallel - Batch Model

Purpose

This example demonstrates how to use the Batch process model to test multiple UUTs simultaneously by running a sequence once for each test socket in the system. The Batch process model generates a batch report that summarizes the results for the UUTs in the batch.

Example File Location

```
<TestStand Public>\Examples\Parallel Testing\Testing UUTs
in Parallel - Batch Model\Testing UUTs in Parallel - Batch
Model.seq
```

Highlighted Features

- Batch process model
- Batch Synchronization step
- Flow Control steps
- Synchronization panel

Major API

None

Prerequisites

None

How to Use This Example

The Set Chamber Temperature step in the `MainSequence` applies an action to the entire UUT batch. The Synchronization panel of the Properties tab of the Step Settings pane for the step sets the Batch Synchronization option to **One thread only** because the sequence needs to execute this action only once for the entire batch.

The Pulse Test step calls the Pulse Test subsequence, which you can select on the Sequences pane. This sequence contains Batch Synchronization steps that create a serial synchronized section around the other steps in the sequence to force all the UUTs in the batch to execute the steps on one UUT before the next UUT can begin. In a real-world application, controlling UUT execution in a batch can be useful in cases in which you have only one set of testing equipment available for certain procedures.

The Frequency Sweep step in the `MainSequence` does not specify any synchronization options, and so all UUTs run this step in parallel.

Select **Execute»Single Pass** to run the sequence and review the behavior of the batch execution. When the test completes, the sequence launches a dialog box that displays the time required to perform the frequency test on all UUTs. The dialog box divides the total time by the number of UUTs and displays the test time per UUT to illustrate how you can increase throughput by testing UUTs in parallel.



Note

- By default, a sequence file uses the Sequential process model, but you can use the Station Model control on the Model tab of the Station Options dialog box to specify that the Batch process model is the default model for all sequence files.
- You can use the Model Options dialog box to configure the number of test sockets you want to use for the batch.

Related concepts:

- [Batch Process Model](#)
- [TestStand Directory Structure](#)
- [Sequential Process Model](#)

Testing UUTs in Parallel - Parallel Model

Purpose

This example demonstrates use of the Parallel process model to test multiple UUTs in parallel. With the Parallel process model, each UUT executes independently, and you can start and stop testing on any test socket at any time.

Example File Location

```
<TestStand Public>\Examples\Parallel Testing\Testing UUTs  
in Parallel - Parallel Model\Testing UUTs in Parallel -  
Parallel Model.seq
```

Highlighted Features

Parallel process model

Major API

None

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequences and steps in the example.

1. Open `Testing UUTs in Parallel - Parallel Model.seq`, select **Edit»Sequence File Properties** and click the Advanced tab. The settings on this tab specify that the Parallel process model executes this sequence file. You can configure any sequence file to use the Parallel process model by opening the file, navigating to the Advanced tab of the Sequence File Properties dialog box, selecting **Require Specific Model** for the Model Option setting, and then selecting **ParallelModel.seq** from the Model File control.
2. On the Sequences pane, select the **MainSequence**. This sequence contains one Sequence Call step and one Numeric Limit Test with an associated LabWindows\CVI code module.
3. On the Sequences pane, select the **PreUUTLoop** sequence. This sequence is a callback from the Parallel process model. Steps in this sequence execute this code only once before any UUTs are tested. The Set Chamber Temperature step simulates a setup operation that must be performed to prepare the test station for UUT testing, but it needs to be performed only once before any tests begin.
4. On the Sequences pane, select the **Pulse Test** sequence. This sequence simulates a test that can execute on only one test socket at a time. In many cases, a test station testing multiple UUTs in parallel must share resources (such as hardware) between test sockets to ensure multiple sockets do not attempt to access the resource at the same time.
5. Observe the Enter Locked Section step at the beginning of the sequence and the Exit Locked Section step at the end of the sequence. These steps use the Lock synchronization object in TestStand to ensure only one test socket executes this sequence at a time.

Complete the following steps to run the example.

1. Select **Execute»Test UUTs** to run the sequence.
2. The Set Chamber Temperature dialog box indicates the simulated chamber heating process. This code executes before any UUTs are tested.

3. The UUT Information dialog box displays once the chamber heating process is complete. Use this dialog box to enter a serial number for the UUT and start or stop the execution for each test socket.
4. Once testing has started on a test socket, dialog boxes indicate the status of the tests. Observe that the Parallel model allows each UUT to start and stop testing independently. However, the Lock steps in the Pulse Test sequence cause that sequence to execute on only one test socket at a time. This behavior demonstrates the ability to synchronize among test sockets when using the Parallel model.

Related concepts:

- [Parallel Process Model](#)
- [TestStand Directory Structure](#)

Examples for the TestStand API

The TestStand API examples demonstrate how to implement common applications of the TestStand API in the TestStand expression language and other programming languages. This directory includes the following example programs.

Accessing Arrays Using API

The <TestStand Public>\Examples\TestStand API\Accessing Arrays Using API directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Accessing Arrays Using API - HTBasic

Purpose

This example demonstrates how to use the TestStand API from an HTBasic code module to access a one-dimensional array, two-dimensional array, and an array of strings in TestStand.

Example File Location

<TestStand Public>\Examples\TestStand API\Accessing Arrays Using API\HTBasic\Accessing Arrays Using API.seq

Highlighted Features

- HTBasic Adapter
- TestStand API

Major API

- PropertyObject.GetValNumberByOffset
- PropertyObject.SetValNumberByOffset
- PropertyObject.GetValStringByOffset
- PropertyObject.SetValStringByOffset

Prerequisites

When you use the demo version of HTBasic, you must configure the HTBasic Adapter to use the HTBasic demo server. Select the **Use HTBasic Runtime Server** option in the HTBasic Adapter Configuration dialog box and specify the path and filename for HTBwin.exe.



Note (64-bit TestStand) 64-bit TestStand does not support the HTBasic Adapter.

How to Use This Example

Open the following ArraySubs .prg file, located in the <TestStand Public>\Examples\TestStand API\Accessing Arrays Using API\HTBasic directory, and review the following defined subroutines and the TestStand API calls used to get and set the TestStand local variables:

- Filllocalarray—Demonstrates how to use the TestStand API from HTBasic to populate an array stored in a TestStand local variable.
- Accessarray—Demonstrates how to use the TestStand API from HTBasic to read

an array stored in a TestStand local variable and display a graph of the data.

- `Fill2darray`—Demonstrates how to use the TestStand API from HTBasic to populate a two-dimensional array stored in a TestStand local variable.
- `Access2darray`—Demonstrates how to use the TestStand API from HTBasic to read a two-dimensional array stored in a TestStand local variable.
- `Fillstrarray`—Demonstrates how to use the TestStand API from HTBasic to populate a string array stored in a TestStand local variable.
- `Accessstrarray`—Demonstrates how to use the TestStand API from HTBasic to read a string array stored in a TestStand local variable.

Complete the following steps to review how TestStand calls the subroutines in the HTBasic PRG file and review the created local variables.

1. Open the `AccessingArrays.seq` example sequence file. On the Variables pane, the `NumericArray`, `TwoDimArray`, and `StrArray` local variables have been created, but do not contain any data.
2. On the Steps pane, select an Action step.
3. On the Step Settings pane, review the settings for the Action step. The step does not pass local variables to or return local variables from the subroutine.
4. Select **Execute»Single Pass** to run the sequence. Use the TestStand debugging tools such as breakpoints, the Variables pane, and the Watch View pane to monitor the local variables and verify that the HTBasic subroutines populate the panes with data.

Related concepts:

- [TestStand Directory Structure](#)
- [Debugging Executions](#)

Accessing Arrays Using API - LabWindows/CVI

Purpose

This example demonstrates how to use the TestStand API from a LabWindows/CVI DLL code module to access a one-dimensional array, two-dimensional array, and an array of strings in TestStand.

Example File Location

<TestStand Public>\Examples\TestStand API\Accessing Arrays Using API\CVI\Accessing Arrays Using API.seq

Highlighted Features

- LabWindows/CVI Adapter
- TestStand API

Major API

- PropertyObject.NewSubProperty
- PropertyObject.GetValVariant
- PropertyObject.SetValVariant

Prerequisites

You must have the LabWindows/CVI Run-Time Engine installed and you must configure the LabWindows/CVI Adapter to execute steps in-process. If you want to use the **Execute Steps in an External Instance of CVI** option, you must have the LabWindows/CVI development environment installed.

How to Use This Example

Open the `AccessArrays.c` source file, located in the <TestStand Public>\Examples\TestStand API\Accessing Arrays Using API\CVI directory, and review the following defined functions and the TestStand API calls used to get and set the TestStand local variables:

- `FillLocalArray`—Demonstrates how to use the TestStand API from LabWindows/CVI to populate an array stored in a TestStand local variable.
- `AccessLocalArray`—Demonstrates how to use the TestStand API from LabWindows/CVI to read an array stored in a TestStand local variable and display a graph of the data.
- `Create2DArray`—Demonstrates how to use the TestStand API from LabWindows/CVI to dynamically create and populate a two-dimensional array stored in a TestStand local variable



Note Normally, this function would use the `InsertIfMissing` property option to create the local variable, but this example does not use this property option to maintain compatibility with previous versions of TestStand.

- `Access2DArray`—Demonstrates how to use the TestStand API from LabWindows/CVI to read a two-dimensional array stored in a TestStand local variable and display a graph of the data.
- `SetStringArray`—Demonstrates how to use the TestStand API from LabWindows/CVI to populate a string array stored in a TestStand local variable.
- `GetStringArray`—Demonstrates how to use the TestStand API from LabWindows/CVI to read a string array stored in a TestStand local variable and display the strings in a message popup.

Complete the following steps to review how TestStand calls the functions in the DLL, review the created local variables, and review the parameters passed to and from the functions.

1. Open the `AccessingArrays.seq` example sequence file. On the Variables pane, the `NumericArray` and `StringArray` local variables have been created, but do not contain any data.
2. On the Steps pane, select an Action step.
3. On the Step Settings pane, review the settings for the Action step. The step does not pass local variables to or return local variables from the function.
4. Select **Execute»Single Pass** to run the sequence. The DLL calls populate the local variables with data, and LabWindows/CVI accesses and displays the data.

Related concepts:

- [TestStand Directory Structure](#)
- [Programming with the TestStand API in LabWindows/CVI](#)

Accessing Arrays Using API - LabVIEW

Purpose

This example demonstrates how to use the TestStand API from a LabVIEW code

module to access a one-dimensional array, two-dimensional array, and an array of strings in TestStand.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Accessing Arrays  
Using API\LabVIEW\Accessing Arrays Using API.seq
```

Highlighted Features

- LabVIEW Adapter
- TestStand API

Major API

- PropertyObject.NewSubProperty
- PropertyObject.SetValVariant

Prerequisites

You must have the LabVIEW development system installed and you must configure the LabVIEW Adapter to use the LabVIEW development system.

How to Use This Example

Open the following example VIs, located in the <TestStand Public>\Examples\TestStand API\Accessing Arrays Using API\LabVIEW directory, and review the TestStand API calls used to get and set the TestStand local variables:

- `FillLocalArray.vi`—Demonstrates how to use the TestStand API from LabVIEW to populate an array stored in a TestStand local variable.
- `AccessArray.vi`—Demonstrates how to use the TestStand API from LabVIEW to read an array stored in a TestStand local variable and display a graph of the data.
- `Create2DArray.vi`—Demonstrates how to use the TestStand API from LabVIEW to dynamically create and populate a two-dimensional array stored in a TestStand local variable.
- `Access2DArray.vi`—Demonstrates how to use the TestStand API from

LabVIEW to read a two-dimensional array stored in a TestStand local variable and display a graph of the data.

- `SetStringArray.vi`—Demonstrates how to use the TestStand API from LabVIEW to populate a string array stored in a TestStand local variable.
- `GetStringArray.vi`—Demonstrates how to use the TestStand API from LabVIEW to read a string array stored in a TestStand local variable and display the strings in the VI.

Complete the following steps to review how TestStand calls the VIs, review the created local variables, and review the parameters passed to and from the VIs.

1. Open the `AccessingArrays.seq` example sequence file. On the Variables pane, the `NumericArray` and `StringArray` local variables have been created, but do not contain any data.
2. On the Steps pane, select an Action step.
3. On the Step Settings pane, review the settings for the Action step. The step does not pass local variables to or return local variables from the VI.
4. Select **Execute»Single Pass** to run the sequence. The VIs populate the local variables with data, and LabVIEW accesses and displays the data.

Related concepts:

- [TestStand Directory Structure](#)
- [Programming with the TestStand API in LabVIEW](#)

Accessing Arrays Using API - MFC

Purpose

This example demonstrates how to use the TestStand API from a Microsoft Foundation Class (MFC) Library code module to access a one-dimensional array, two-dimensional array, and an array of strings in TestStand.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Accessing Arrays  
Using API\MFC\Accessing Arrays Using API.seq
```


Highlighted Features

- C/C++ DLL Adapter
- TestStand API

Major API

- `PropertyObject.NewSubProperty`
- `PropertyObject.GetValVariant`
- `PropertyObject.SetValVariant`

Prerequisites

None

How to Use This Example

AppWizard creates `AccessingArrays.dll`, which demonstrates the basics of using the Microsoft Foundation Class (MFC) Library and serves as a starting point for writing a custom DLL.

The following files, located in the `<TestStand Public>\Examples\TestStand API\Accessing Arrays Using API\MFC` directory, make up `AccessingArrays.dll`:

- `AccessingArrays.h`—The main header file that declares the `CAccessingArraysApp` class for the DLL.
- `AccessingArrays.cpp`—The main DLL source file that contains the `CAccessingArraysApp` class.
- `AccessingArrays.rc`—A list of all the Microsoft Windows resources the program uses, including icons, bitmaps, and cursors stored in the `res` subdirectory. You can directly edit this file in Microsoft Developer Studio.
- `res\AccessingArrays.rc2`—Contains resources Developer Studio does not edit. Place all resources the resource editor cannot edit in this file.
- `AccessingArrays.def`—Contains DLL information that must be provided to run with Windows. This file defines parameters such as the name and description of the DLL and exports functions from the DLL.

- `AccessingArrays.clw`—Contains information ClassWizard uses to edit existing classes, add new classes, and store information required to create and edit message maps and dialog data maps and to create prototype member functions.

Other standard files located in the `<TestStand Public>\Examples\AccessingArraysUsingAPI\UsingMFC` directory include the following:

- `StdAfx.h` and `StdAfx.cpp`—Used to build a precompiled header (PCH) file named `AccessingArrays.pch` and a precompiled types file named `StdAfx.obj`.
- `Resource.h`—The standard header file that defines new resource IDs. Developer Studio reads and updates this file.



Note AppWizard uses `TODO:` to indicate parts of the source code to which you must add or that you must customize.

Related concepts:

- [TestStand Directory Structure](#)
- [Organizing Test Program Files with LabVIEW-Built Shared Libraries](#)

Accessing Arrays Using API - .NET

Purpose

This example demonstrates how to use the TestStand API from a .NET assembly code module to access a one-dimensional array, two-dimensional array, and an array of strings in TestStand.

Example File Location

`<TestStand Public>\Examples\TestStand API\Accessing Arrays Using API\DotNet\Accessing Arrays Using API.seq`

Highlighted Features

- .NET Adapter

- TestStand API

Major API

- PropertyObject.NewSubProperty
- PropertyObject.GetValVariant
- PropertyObject.SetValVariant

Prerequisites

None

How to Use This Example

Complete the following steps to review how TestStand calls the functions in the DLL, review the created local variables, and review the parameters passed to and from the functions.

1. Open the `AccessingArraysDotNet.seq` example sequence file. On the Variables pane, the `NumericArray` and `StringArray` local variables have been created, but do not contain any data.
2. On the Steps pane, select an Action step.
3. On the Step Settings pane, review the settings for the Action step. The step does not pass local variables to or return local variables from the function.
4. Select **Execute»Single Pass** to run the sequence. The DLL calls populate the local variables with data, and the .NET assemblies access and display the data.

Related concepts:

- [TestStand Directory Structure](#)

Accessing Properties Using API

The `<TestStand Public>\Examples\TestStand API\Accessing Properties Using API` directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Accessing Properties Using API - LabWindows/CVI

Purpose

This example shows how to access built-in and dynamic TestStand properties. Built-in TestStand properties are defined by the TestStand API, and are available for any object of a given class. Dynamic properties, also known as SubProperties, can vary between objects of the same class.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Accessing  
Properties Using API\CVI\Accessing Properties Using API.seq
```

Highlighted Features

- Built-In Properties
- Dynamic Properties
- RunState Property
- TestStand API

Major API

- PropertyObject.GetValNumber
- PropertyObject.GetValString
- PropertyObject.SetValNumber
- PropertyObject.SetValBoolean
- PropertyObject.SetValVariant

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. As the example executes, read the message popups for information about built-in and dynamic properties and how they are accessed.

Complete the following steps to review the sequences and steps in this example.

1. Open the **Accessing Built-In Properties** sequence from the Sequences pane. Select the **Access Built-in properties** step, then click **Edit Code** in the Module tab of the Step Settings pane to open the code module. This code module uses the TestStand API to access built-in properties and methods.
2. Open the **Accessing Dynamic Properties** sequence from the Sequences pane. Select the **Get Step Results** step, then click **Edit Code** in the Module tab of the Step Settings pane to open the code module. This code module uses the TestStand PropertyObject methods to get the values of dynamic step properties.
3. Select the **Set Step Results** step, then click **Edit Code** in the Module tab of the Step Settings pane to open the code module. This code module uses the TestStand PropertyObject methods to set the values of dynamic step properties.
4. Select the **Set Variables** step, then click **Edit Code** in the Module tab of the Step Settings pane to open the code module. This code module uses the TestStand PropertyObject methods to set the value of a local variable, and create a new local variable that exists only at run-time.

Related concepts:

- [Accessing Built-in Properties](#)
- [Accessing Dynamic Properties](#)
- [TestStand Directory Structure](#)
- [RunState Subproperties](#)

Accessing Properties Using API - LabVIEW

Purpose

This example shows how to access built-in and dynamic TestStand properties. Built-in TestStand properties are defined by the TestStand API, and are available for any object of a given class. Dynamic properties, also known as SubProperties, can vary between objects of the same class.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Accessing  
Properties Using API\LabVIEW\Accessing Properties Using  
API.seq
```

Highlighted Features

- Built-In Properties
- Dynamic Properties
- RunState Property
- TestStand API

Major API

- PropertyObject.GetValNumber
- PropertyObject.GetValString
- PropertyObject.SetValNumber
- PropertyObject.SetValBoolean
- PropertyObject.SetValVariant

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. As the example executes, read the message popups for information about built-in and dynamic properties and how they are accessed.

Complete the following steps to review the sequences and steps in this example.

1. Open the **Accessing Built-In Properties** sequence from the Sequences pane. Select the **Access Built-in properties** step, then click **Edit VI** in the Module tab of the Step Settings pane to open the code module. This code module uses the TestStand API

to access built-in properties and methods.

2. Open the **Accessing Dynamic Properties** sequence from the Sequences pane. Select the **Get Step Results** step, then click **Edit VI** in the Module tab of the Step Settings pane to open the code module. This code module uses the TestStand PropertyObject methods to get the values of dynamic step properties.
3. Select the **Set Step Results** step, then click **Edit VI** in the Module tab of the Step Settings pane to open the code module. This code module uses the TestStand PropertyObject methods to set the values of dynamic step properties.
4. Select the **Set Variables** step, then click **Edit VI** in the Module tab of the Step Settings pane to open the code module. This code module uses the TestStand PropertyObject methods to set the value of a local variable, and create a new local variable that exists only at run-time.

Related concepts:

- [Accessing Built-in Properties](#)
- [Accessing Dynamic Properties](#)
- [TestStand Directory Structure](#)
- [RunState Subproperties](#)

Accessing Properties Using API - MFC

None

Purpose

This example shows how to access built-in and dynamic TestStand properties. Built-in TestStand properties are defined by the TestStand API, and are available for any object of a given class. Dynamic properties, also known as SubProperties, can vary between objects of the same class.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Accessing
Properties Using API\MFC\Accessing Properties Using API.seq
```

Highlighted Features

- Built-In Properties
- Dynamic Properties
- RunState Property
- TestStand API

Major API

- PropertyObject.GetValNumber
- PropertyObject.GetValString
- PropertyObject.SetValNumber
- PropertyObject.SetValBoolean
- PropertyObject.SetValVariant

Prerequisites

None

How to Use This Example

AppWizard creates `AccessingPropertiesAndVariables.dll`, which demonstrates the basics of using the Microsoft Foundation Class (MFC) Library and serves as a starting point for writing a custom DLL.

The following files, located in the `<TestStand Public>\Examples\TestStand API\Accessing Properties Using API\MFC` directory, make up `AccessingPropertiesAndVariables.dll`:

- `AccessingPropertiesAndVariables.h`—The main header file that declares the `CAccessingPropertiesAndVariablesApp` class for the DLL.
- `AccessingPropertiesAndVariables.cpp`—The main DLL source file that contains the `CAccessingPropertiesAndVariablesApp` class.
- `AccessingPropertiesAndVariables.rc`—A list of all the Microsoft Windows resources the program uses, including icons, bitmaps, and cursors stored in the `res` subdirectory. You can directly edit this file in Microsoft Developer Studio.

- `res\AccessingPropertiesAndVariables.rc2`—Contains resources Developer Studio does not edit. Place all resources the resource editor cannot edit in this file.
- `AccessingPropertiesAndVariables.def`—Contains DLL information that must be provided to run with Windows. This file defines parameters such as the name and description of the DLL and exports functions from the DLL.
- `AccessingPropertiesAndVariables.clw`—Contains information ClassWizard uses to edit existing classes, add new classes, and store information required to create and edit message maps and dialog data maps and to create prototype member functions.

Other standard files located in the `<TestStand Public>\Examples\Accessing Properties Using API\MFC` directory include the following:

- `StdAfx.h` and `StdAfx.cpp`—Used to build a precompiled header (PCH) file named `AccessingPropertiesAndVariables.pch` and a precompiled types file named `StdAfx.obj`.
- `Resource.h`—The standard header file that defines new resource IDs. Developer Studio reads and updates this file.



Note AppWizard uses `TODO:` to indicate parts of the source code to which you must add or that you must customize.

Related concepts:

- [TestStand Directory Structure](#)

Accessing Properties Using API - .NET

Purpose

This example shows how to access built-in and dynamic TestStand properties. Built-in TestStand properties are defined by the TestStand API, and are available for any object of a given class. Dynamic properties, also known as SubProperties, can vary between objects of the same class.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Accessing  
Properties Using API\DotNet\Accessing Properties Using  
API.seq
```

Highlighted Features

- Built-In Properties
- Dynamic Properties
- RunState Property
- TestStand API

Major API

- PropertyObject.GetValNumber
- PropertyObject.GetValString
- PropertyObject.SetValNumber
- PropertyObject.SetValBoolean
- PropertyObject.SetValVariant

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. As the example executes, read the message popups for information about built-in and dynamic properties and how they are accessed.

Complete the following steps to review the sequences and steps in this example.

1. Open the Accessing Built-In Properties sequence from the Sequences pane. Select the **Access Built-in properties** step, then click **Edit In Visual Studio** in the Module tab of the Step Settings pane to open the code module. This code module uses the

TestStand API to access built-in properties and methods.

2. Open the Accessing Dynamic Properties sequence from the Sequences pane. Select the **Get Step Results** step, then click **Edit In Visual Studio** in the Module tab of the Step Settings pane to open the code module. This code module uses the TestStand PropertyObject methods to get the values of dynamic step properties.
3. Select the **Set Step Results** step, then click **Edit In Visual Studio** in the Module tab of the Step Settings pane to open the code module. This code module uses the TestStand PropertyObject methods to set the values of dynamic step properties.
4. Select the **Set Variables** step, then click **Edit In Visual Studio** in the Module tab of the Step Settings pane to open the code module. This code module uses the TestStand PropertyObject methods to set the value of a local variable, and create a new local variable that exists only at run-time.

Related concepts:

- [Accessing Built-in Properties](#)
- [Accessing Dynamic Properties](#)
- [TestStand Directory Structure](#)
- [RunState Subproperties](#)

Accessing Properties Using API - TestStand Expressions

Purpose

This example shows how to access built-in and dynamic TestStand properties. Built-in TestStand properties are defined by the TestStand API, and are available for any object of a given class. Dynamic properties, also known as SubProperties, can vary between objects of the same class.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Accessing  
Properties Using API\TestStand Expressions\Accessing  
Properties Using API.seq
```

Highlighted Features

- Built-In Properties

- Dynamic Properties
- RunState Property
- TestStand API

Major API

- PropertyObject.GetValNumber
- PropertyObject.GetValString
- PropertyObject.SetValNumber
- PropertyObject.SetValBoolean
- PropertyObject.SetValVariant

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. As the example executes, read the message popups for information about built-in and dynamic properties and how they are accessed.

Complete the following steps to review the sequences and steps in this example.

1. Open the **Accessing Built-In Properties** sequence from the Sequences pane. Review the statement steps in this sequence. These steps access built-in properties and methods using the TestStand API.
2. Open the **Accessing Dynamic Properties** sequence from the Sequences pane. Review the first set of statement steps under the Get Step Properties label. These steps use the TestStand PropertyObject methods to get the values of dynamic step properties.
3. Review the set of statement steps under the Set Step Properties label. These steps use the TestStand PropertyObject methods to set the values of dynamic step properties.
4. Review the set of statement steps under the Get Step Properties label. These steps use the TestStand PropertyObject methods to set the value of a local variable, and

dynamically create a local variable that exists only at run-time.

Related concepts:

- [Accessing Built-in Properties](#)
- [Accessing Dynamic Properties](#)
- [TestStand Directory Structure](#)
- [RunState Subproperties](#)

Building a Sequence Using API

The <TestStand Public>\Examples\TestStand API\Building a Sequence Using API directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Building a Sequence Using API - LabVIEW

Purpose

This example demonstrates how to programmatically create and execute new test sequences. This example uses an XML file to create new steps and insert them into a new sequence. The example then saves and executes the new sequence file.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Building a  
Sequence Using API\LabVIEW\Building a Sequence Using API -  
XML.seq
```

Highlighted Features

- TestStand API
- Flow Control steps

Major API

- Engine.NewSequenceFile
- Engine.NewStep
- Sequence.InsertStep
- Engine.GetSequenceFileEx
- Engine.NewExecution
- Execution.WaitForEndEx
- Engine.ReleaseSequenceFileEx

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to start the example.
2. When prompted, select the example XML file. The data in this file will be used to create a new sequence file.
3. Notice that a new execution is created. This execution is running the dynamically created sequence file.
4. Open the sample XML file located in <TestStand Public>\TestStand API\Building a Sequence Using API\Common in a text editor. You can modify this file to change the steps in the generated sequence file.

Related concepts:

- [TestStand Directory Structure](#)

Building a Sequence Using API - .NET

Purpose

This example demonstrates how to programmatically create and execute new test sequences. This example uses an XML file to create new steps and insert them into a new sequence. The example then saves and executes the new sequence file.

Example File Location

<TestStand Public>\Examples\TestStand API\Building a Sequence Using API\DotNet\Building a Sequence Using API - XML.seq

Highlighted Features

- TestStand API
- Flow Control steps

Major API

- Engine.NewSequenceFile
- Engine.NewStep
- Sequence.InsertStep
- Engine.GetSequenceFileEx
- Engine.NewExecution
- Execution.WaitForEndEx
- Engine.ReleaseSequenceFileEx

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to start the example.
2. When prompted, select the example XML file. The data in this file will be used to create a new sequence file.
3. Notice that a new execution is created. This execution is running the dynamically created sequence file.
4. Open the sample XML file located in <TestStand Public>\TestStand API\Building a Sequence Using API\Common in a text editor. You can modify this file to change the steps in the generated sequence file.

Related concepts:

- [TestStand Directory Structure](#)

Building a Sequence Using API - TestStand Expressions (INI)

Purpose

This example demonstrates how to programmatically create and execute new test sequences. This example uses an INI file to create new steps and insert them into a new sequence. The example then saves and executes the new sequence file.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Building a  
Sequence Using API\TestStand Expressions\Building a  
Sequence Using API - INI.seq
```

Highlighted Features

- TestStand API
- Flow Control steps

Major API

- Engine.NewSequenceFile
- Engine.NewStep
- Sequence.InsertStep
- Engine.GetSequenceFileEx
- Engine.NewExecution
- Execution.WaitForEndEx
- Engine.ReleaseSequenceFileEx

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to start the example.
2. When prompted, select the example INI file. The data in this file will be used to create a new sequence file.
3. Notice that a new execution is created. This execution is running the dynamically created sequence file.
4. Open the sample INI file located in `<TestStand Public>\TestStand API\Building a Sequence Using API\Common` in a text editor. You can modify this file to change the steps in the generated sequence file.

Related concepts:

- [TestStand Directory Structure](#)

Building a Sequence Using API - TestStand Expressions (XML)

Purpose

This example demonstrates how to programmatically create and execute new test sequences. This example uses an XML file to create new steps and insert them into a new sequence. The example then saves and executes the new sequence file.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Building a  
Sequence Using API\TestStand Expressions\Building a  
Sequence Using API - XML.seq
```

Highlighted Features

- TestStand API
- Flow Control steps

Major API

- Engine.NewSequenceFile
- Engine.NewStep
- Sequence.InsertStep
- Engine.GetSequenceFileEx
- Engine.NewExecution
- Execution.WaitForEndEx
- Engine.ReleaseSequenceFileEx

Prerequisites

None

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to start the example.
2. When prompted, select the example XML file. The data in this file will be used to create a new sequence file.
3. Notice that a new execution is created. This execution is running the dynamically created sequence file.
4. Open the sample XML file located in <TestStand Public>\TestStand API\Building a Sequence Using API\Common in a text editor. You can modify this file to change the steps in the generated sequence file.

Related concepts:

- [TestStand Directory Structure](#)

Creating and Deleting Users Using API

Purpose

This example demonstrates how to programmatically create and delete users.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Creating &
Deleting Users Using API\Creating & Deleting Users Using
API.seq
```

Highlighted Features

- User management

Major API

- Engine.GetUserGroup
- Engine.NewUser
- Engine.UserNameExists
- Engine.UsersFile
- PropertyObject.SetPropertyObjectByOffset
- PropertyObject.SetValStringByOffset
- UsersFile.UserList

Prerequisites

You must be logged into TestStand as a user with administrator privileges.

How to Use This Example

Complete the following steps to review the sequences and steps in this example.

1. On the Steps pane of the `MainSequence`, select the **Create or Delete a User** step.
2. On the Step Settings pane, click the Text and Buttons tab. When you run the sequence, you select whether you want to create a user or delete a user. The If and Else steps call the Create User sequence or Delete User sequence, depending on which option you select.
3. On the Sequences pane, select the **Create User** sequence, which specifies the following functionality:
 - a. The Get New User Name step prompts for a new user name.
 - b. The If step uses the `Engine.UserNameExists` method to determine whether the user name already exists. If the user name does not exist, the user is created.

- c. The Select User Group step in the If structure prompts you to select a group for the user.
 - d. A series of Statement steps perform API calls to create the new user and increment the change count of the user file.
4. On the Steps pane, select each Statement step one at a time and click the Expression edit tab on the Step Settings pane to review the expression each step uses. The steps use the `Engine.NewUser` method to create the new user, the `UsersFile.UserList` property and `PropertyObject.SetPropertyObjectByOffset` method to insert the new user in the user list, and the `Engine.GetUserGroup` method and `PropertyObject.SetValStringByOffset` method to add the user to a group.
5. On the Sequences pane, select the **Delete User** sequence, which specifies the following functionality:
 - a. The Get User Names step, which is a Sequence Call step, calls a subsequence to generate an array of all user names in the user list.
 - b. In the If structure, the Delete User sequence verifies that more than one user exists so that you cannot delete the only user and become locked out of TestStand.
 - c. The Delete User sequence stores the user name to delete in a local variable before the user is actually deleted and removed from any groups.
 - d. A Statement step increments the change count of the user file.
6. On the Steps pane of the `MainSequence`, select the Save Users File step.
7. On the Step Settings pane, click the **Text and Buttons** tab. When you run the sequence and have selected whether to create a new user or delete a user, the precondition evaluates whether the Users file has been modified. If the file has been modified, the step prompts you to save the changes. If you select **No**, TestStand reflects the changes in the User Manager but does not save the changes.

Complete the following steps to run the example.

1. On the Sequences pane, select the `MainSequence`.
2. Select **Execute»Run MainSequence** to run the sequence.
3. In the Create or Delete a User dialog box, click **Create User**.
4. In the New User Name dialog box, enter a name for the new user and click **OK**.
5. In the User Group dialog box, select any group for the new user.
6. In the Save the Modified Users File dialog box, select whether you want to save the changes made to the User file.

7. Select **Execute»Restart** to run the sequence again.
8. In the Create or Delete a User dialog box, click **Delete User**.
9. In the Select User dialog box, select the new user you created in step 4.
10. In the Save the Modified Users File dialog box, select whether you want to save the changes made to the User file.

Related concepts:

- [TestStand Directory Structure](#)
- [Managing Users](#)
- [Privileges](#)

Creating New Properties Using API

The <TestStand Public>\Examples\TestStand API\Creating New Properties Using API directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Creating New Properties Using API - LabWindows/CVI

Purpose

This example demonstrates how to use the TestStand API to create and assign values to PropertyObjects. With the exception of the TestStand Engine, any object in TestStand can be accessed as a PropertyObject. This arrangement allows the methods demonstrated in this example to be used with many objects in TestStand, including step properties and variables.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Creating New
Properties Using API\CVI\Creating New Properties Using
API.seq
```

Highlighted Features

- TestStand API

Major API

- Engine.NewPropertyObject
- PropertyObject.NewSubProperty
- PropertyObject.InsertSubProperty
- PropertyObject.SetValString
- PropertyObject.SetValNumber
- PropertyObject.SetValBoolean
- PropertyObject.SetValVariant
- PropertyObject.SetFlags

Prerequisites

Report generation should be enabled in order to see the result of adding a PropertyObject to the Result container of a step.

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to run the sequence.
2. TestStand will display Message Popups to explain each section of the example. After each section completes, the execution will automatically break to allow you to observe the results of each approach.
3. When you are ready to continue execution, simply click the green **Resume** button on the debug toolbar.
4. After execution completes, examine the report to observe the results of the final section, where a PropertyObject named SpecialResults was added to a step's Result container.

Complete the following steps to review the sequences and steps in this example:

1. On the Sequences pane, select the **MainSequence**. The MainSequence contains four Sequence Call steps corresponding to the different approaches demonstrated

in this example.

2. On the Sequences pane, select the **Create New PropertyObject** sequence. This sequence uses the `Engine.NewPropertyObject` and `PropertyObject.InsertSubProperty` API calls to create a new `PropertyObject` and insert it into the Locals container as a new Local variable. Although this method of creating `PropertyObjects` is less common than other approaches, it is included for completeness.
3. On the Sequences pane, select the **SetVal Methods** sequence. This sequence uses the `PropertyObject.SetValString`, `PropertyObject.SetValNumber`, and `PropertyObject.SetValBoolean` methods to create `PropertyObjects` and insert them as Local variables. This approach allows you to create and assign a value to a `PropertyObject` with a single API call.
4. On the Sequences pane, select the **Create New Container** sequence. This sequence demonstrates a method of creating complex container structures with the `SetVal` methods. The `PropertyObject.NewSubProperty` method is used to create a container field for error data, which is a defined type in the TestStand Types pane. Additionally, this sequence uses a `PropertyObject.NewSubProperty` method call to create an array and assigns a value to that array using the `PropertyObject.SetValVariant` method.
5. On the Sequences pane, select the **Insert PropertyObject Into Step Results** sequence. This sequence creates a new `PropertyObject` and inserts it into the step's Result container. The `PropertyObject.SetFlags` TestStand API method is used to mark this `PropertyObject` to be included in the report generated after execution. This demonstrates a common use case for creating new `PropertyObjects` in a test sequence.

Related concepts:

- [TestStand Directory Structure](#)

Creating New Properties Using API - LabVIEW

Purpose

This example demonstrates how to use the TestStand API to create and assign values to `PropertyObjects`. With the exception of the TestStand Engine, any object in TestStand can be accessed as a `PropertyObject`. This arrangement allows the methods

demonstrated in this example to be used with many objects in TestStand, including step properties and variables.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Creating New  
Properties Using API\LabVIEW\Creating New Properties Using  
API.seq
```

Highlighted Features

- TestStand API

Major API

- Engine.NewPropertyObject
- PropertyObject.NewSubProperty
- PropertyObject.InsertSubProperty
- PropertyObject.SetValString
- PropertyObject.SetValNumber
- PropertyObject.SetValBoolean
- PropertyObject.SetValVariant
- PropertyObject.SetFlags

Prerequisites

Report generation should be enabled in order to see the result of adding a PropertyObject to the Result container of a step.

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to run the sequence.
2. TestStand will display Message Popups to explain each section of the example. After each section completes, the execution will automatically break to allow you to observe the results of each approach.
3. When you are ready to continue execution, simply click the green **Resume** button

on the debug toolbar.

4. After execution completes, examine the report to observe the results of the final section, where a `PropertyObject` named `SpecialResults` was added to a step's Result container.

Complete the following steps to review the sequences and steps in this example:

1. On the Sequences pane, select the **MainSequence**. The `MainSequence` contains four Sequence Call steps corresponding to the different approaches demonstrated in this example.
2. On the Sequences pane, select the **Create New PropertyObject** sequence. This sequence uses the `Engine.NewPropertyObject` and `PropertyObject.InsertSubProperty` API calls to create a new `PropertyObject` and insert it into the Locals container as a new Local variable. Although this method of creating `PropertyObjects` is less common than other approaches, it is included for completeness.
3. On the Sequences pane, select the **SetVal Methods** sequence. This sequence uses the `PropertyObject.SetValString`, `PropertyObject.SetValNumber`, and `PropertyObject.SetValBoolean` methods to create `PropertyObjects` and insert them as Local variables. This approach allows you to create and assign a value to a `PropertyObject` with a single API call.
4. On the Sequences pane, select the **Create New Container** sequence. This sequence demonstrates a method of creating complex container structures with the `SetVal` methods. The `PropertyObject.NewSubProperty` method is used to create a container field for error data, which is a defined type in the TestStand Types pane. Additionally, this sequence uses a `PropertyObject.NewSubProperty` method call to create an array and assigns a value to that array using the `PropertyObject.SetValVariant` method.
5. On the Sequences pane, select the **Insert PropertyObject Into Step Results** sequence. This sequence creates a new `PropertyObject` and inserts it into the step's Result container. The `PropertyObject.SetFlags` TestStand API method is used to mark this `PropertyObject` to be included in the report generated after execution. This demonstrates a common use case for creating new `PropertyObjects` in a test sequence.

Related concepts:

- [TestStand Directory Structure](#)

Creating New Properties Using API - .NET

Purpose

This example demonstrates how to use the TestStand API to create and assign values to PropertyObjects. With the exception of the TestStand Engine, any object in TestStand can be accessed as a PropertyObject. This arrangement allows the methods demonstrated in this example to be used with many objects in TestStand, including step properties and variables.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Creating New  
Properties Using API\DotNet\Creating New Properties Using  
API.seq
```

Highlighted Features

- TestStand API

Major API

- Engine.NewPropertyObject
- PropertyObject.NewSubProperty
- PropertyObject.InsertSubProperty
- PropertyObject.SetValString
- PropertyObject.SetValNumber
- PropertyObject.SetValBoolean
- PropertyObject.SetValVariant
- PropertyObject.SetFlags

Prerequisites

Report generation should be enabled in order to see the result of adding a

PropertyObject to the Result container of a step.

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to run the sequence.
2. TestStand will display Message Popups to explain each section of the example. After each section completes, the execution will automatically break to allow you to observe the results of each approach.
3. When you are ready to continue execution, simply click the green **Resume** button on the debug toolbar.
4. After execution completes, examine the report to observe the results of the final section, where a PropertyObject named SpecialResults was added to a step's Result container.

Complete the following steps to review the sequences and steps in this example:

1. On the Sequences pane, select the **MainSequence**. The MainSequence contains four Sequence Call steps corresponding to the different approaches demonstrated in this example.
2. On the Sequences pane, select the **Create New PropertyObject** sequence. This sequence uses the `Engine.NewPropertyObject` and `PropertyObject.InsertSubProperty` API calls to create a new PropertyObject and insert it into the Locals container as a new Local variable. Although this method of creating PropertyObjects is less common than other approaches, it is included for completeness.
3. On the Sequences pane, select the **SetVal Methods** sequence. This sequence uses the `PropertyObject.SetValString`, `PropertyObject.SetValNumber`, and `PropertyObject.SetValBoolean` methods to create PropertyObjects and insert them as Local variables. This approach allows you to create and assign a value to a PropertyObject with a single API call.
4. On the Sequences pane, select the **Create New Container** sequence. This sequence demonstrates a method of creating complex container structures with the SetVal methods. The `PropertyObject.NewSubProperty` method is used to create a container field for error data, which is a defined type in the TestStand Types pane. Additionally, this sequence uses a `PropertyObject.NewSubProperty`

method call to create an array and assigns a value to that array using the `>PropertyObject.SetValVariant` method.

5. On the Sequences pane, select the **Insert PropertyObject Into Step Results** sequence. This sequence creates a new PropertyObject and inserts it into the step's Result container. The PropertyObject.SetFlags TestStand API method is used to mark this PropertyObject to be included in the report generated after execution. This demonstrates a common use case for creating new PropertyObjects in a test sequence.

Related concepts:

- [TestStand Directory Structure](#)

Creating New Properties Using API - TestStand Expressions

Purpose

This example demonstrates how to use the TestStand API to create and assign values to PropertyObjects. With the exception of the TestStand Engine, any object in TestStand can be accessed as a PropertyObject. This arrangement allows the methods demonstrated in this example to be used with many objects in TestStand, including step properties and variables.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Creating New
Properties Using API\TestStand Expressions\Creating New
Properties Using API.seq
```

Highlighted Features

- TestStand API

Major API

- Engine.NewPropertyObject
- PropertyObject.NewSubProperty
- PropertyObject.InsertSubProperty

- `PropertyObject.SetValString`
- `PropertyObject.SetValNumber`
- `PropertyObject.SetValBoolean`
- `PropertyObject.SetValVariant`
- `PropertyObject.SetFlags`

Prerequisites

Report generation should be enabled in order to see the result of adding a `PropertyObject` to the Result container of a step.

How to Use This Example

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to run the sequence.
2. TestStand will display Message Popups to explain each section of the example. After each section completes, the execution will automatically break to allow you to observe the results of each approach.
3. When you are ready to continue execution, simply click the green **Resume** button on the debug toolbar.
4. After execution completes, examine the report to observe the results of the final section, where a `PropertyObject` named `SpecialResults` was added to a step's Result container.

Complete the following steps to review the sequences and steps in this example:

1. On the Sequences pane, select the **MainSequence**. The `MainSequence` contains four Sequence Call steps corresponding to the different approaches demonstrated in this example.
2. On the Sequences pane, select the **Create New PropertyObject** sequence. This sequence uses the `Engine.NewPropertyObject` and `PropertyObject.InsertSubProperty` API calls to create a new `PropertyObject` and insert it into the Locals container as a new Local variable. Although this method of creating `PropertyObjects` is less common than other approaches, it is included for completeness.
3. On the Sequences pane, select the **SetVal Methods** sequence. This sequence uses the `PropertyObject.SetValString`,

`PropertyObject.SetValNumber`, and

`PropertyObject.SetValBoolean` methods to create `PropertyObject`s and insert them as Local variables. This approach allows you to create and assign a value to a `PropertyObject` with a single API call.

4. On the Sequences pane, select the **Create New Container** sequence. This sequence demonstrates a method of creating complex container structures with the `SetVal` methods. The `PropertyObject.NewSubProperty` method is used to create a container field for error data, which is a defined type in the TestStand Types pane. Additionally, this sequence uses a `PropertyObject.NewSubProperty` method call to create an array and assigns a value to that array using the `PropertyObject.SetValVariant` method.
5. On the Sequences pane, select the **Insert PropertyObject Into Step Results** sequence. This sequence creates a new `PropertyObject` and inserts it into the step's Result container. The `PropertyObject.SetFlags` TestStand API method is used to mark this `PropertyObject` to be included in the report generated after execution. This demonstrates a common use case for creating new `PropertyObject`s in a test sequence.

Related concepts:

- [TestStand Directory Structure](#)

Executing Sequences Using API

The `<TestStand Public>\Examples\TestStand API\Executing Sequences Using API` directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Executing Sequences Using API - LabWindows/CVI

Purpose

This example demonstrates various methods of calling a sequence using the TestStand API.



Note If you are developing a TestStand user interface which must handle TestStand UI messages, do not use the `WaitForEndEx()` method, since this method does not process the UI messages while it waits for the execution to complete. If you need to call this method from a user interface, you must call this method from a non-UI thread.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Executing  
Sequences Using API\CVI\Executing Sequences Using API.seq
```

Highlighted Features

- TestStand API

Major API

- `Engine.GetSequenceFileEx`
- `Engine.NewExecution`
- `Engine.ReleaseSequenceFileEx`

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequences and steps in the example.

1. Open **Executing Sequences Using API.seq**.
2. On the Sequences pane, select the **MainSequence**. The `MainSequence` contains four Sequence Call steps that correspond to the different approaches demonstrated in this example.
3. On the Sequences pane, select the **Execute With No Process Model** sequence. This sequence uses the `Engine.GetSequenceFileEx` and `Engine.NewExecution` methods from the TestStand API to open and execute a sequence file. Because this type of execution does not use a process model, it is

similar to selecting **Execute»Run MainSequence** in the target sequence file.

4. After execution completes, the `Engine.ReleaseSequenceFileEx` method releases the target sequence file. This is a necessary step for any sequence file loaded with the `Engine.GetSequenceFile` method.
5. On the Sequences pane, select the **Execute With Process Model** sequence. This sequence loads a process model with the `Engine.GetSequenceFileEx` method and uses the `Engine.NewExecution` method to execute a client sequence file using one of the entry points from the process model.
6. On the Sequences pane, select the **Execute With Sequence Parameters** sequence. This sequence passes parameters to the target sequence by using the `Engine.NewPropertyObject`, `PropertyObject.SetValNumber`, and `PropertyObject.SetValString` methods to create a container to store the parameters. The parameters must be declared in the same order they appear in the target sequence. Observe that the new `PropertyObject` container is passed to the `Engine.NewExecution` method as one of its parameters.
7. Open **Target Sequence File.seq**.
8. On the Sequences pane, select the **MainSequence**. This sequence includes a Message Popup step to indicate to the user that the sequence has executed.
9. On the Sequences pane, select the **Subsequence**. Observe that this sequence includes two parameters, which are displayed in the Message Popup step.

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to run the sequence.
2. As each section of code executes, the Message Popup from the target sequence appears. Click **OK** on these dialog boxes to continue execution.
3. When the `Execute With Sequence Parameters` sequence executes, enter a number and string in the Message Popups that appear. Notice that these values are displayed in the final Message Popup.

Related concepts:

- [TestStand Directory Structure](#)

Executing Sequences Using API - LabVIEW

Purpose

This example demonstrates various methods of calling a sequence using the TestStand API.



Note If you are developing a TestStand user interface which must handle TestStand UI messages, do not use the `WaitForEndEx()` method, since this method does not process the UI messages while it waits for the execution to complete. If you need to call this method from a user interface, you must call this method from a non-UI thread.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Executing  
Sequences Using API\LabVIEW\Executing Sequences Using  
API.seq
```

Highlighted Features

- TestStand API

Major API

- `Engine.GetSequenceFileEx`
- `Engine.NewExecution`
- `Engine.ReleaseSequenceFileEx`

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequences and steps in the example.

1. Open **Executing Sequences Using API.seq**.
2. On the Sequences pane, select the **MainSequence**. The `MainSequence` contains four Sequence Call steps that correspond to the different approaches demonstrated in this example.
3. On the Sequences pane, select the **Execute With No Process Model** sequence. This sequence uses the `Engine.GetSequenceFileEx` and `Engine.NewExecution` methods from the TestStand API to open and execute a sequence file. Because this type of execution does not use a process model, it is similar to selecting **Execute»Run MainSequence** in the target sequence file.
4. After execution completes, the `Engine.ReleaseSequenceFileEx` method releases the target sequence file. This is a necessary step for any sequence file loaded with the `Engine.GetSequenceFile` method.
5. On the Sequences pane, select the **Execute With Process Model** sequence. This sequence loads a process model with the `Engine.GetSequenceFileEx` method and uses the `Engine.NewExecution` method to execute a client sequence file using one of the entry points from the process model.
6. On the Sequences pane, select the **Execute With Sequence Parameters** sequence. This sequence passes parameters to the target sequence by using the `Engine.NewPropertyObject`, `PropertyObject.SetValNumber`, and `PropertyObject.SetValString` methods to create a container to store the parameters. The parameters must be declared in the same order they appear in the target sequence. Observe that the new `PropertyObject` container is passed to the `Engine.NewExecution` method as one of its parameters.
7. Open **Target Sequence File.seq**.
8. On the Sequences pane, select the **MainSequence**. This sequence includes a Message Popup step to indicate to the user that the sequence has executed.
9. On the Sequences pane, select the **Subsequence**. Observe that this sequence includes two parameters, which are displayed in the Message Popup step.

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to run the sequence.
2. As each section of code executes, the Message Popup from the target sequence appears. Click **OK** on these dialog boxes to continue execution.
3. When the **Execute With Sequence Parameters** sequence executes, enter a number and string in the Message Popups that appear. Notice that these values

are displayed in the final Message Popup.

Related concepts:

- [TestStand Directory Structure](#)

Executing Sequences Using API - .NET

Purpose

This example demonstrates various methods of calling a sequence using the TestStand API.



Note If you are developing a TestStand user interface which must handle TestStand UI messages, do not use the `WaitForEndEx()` method, since this method does not process the UI messages while it waits for the execution to complete. If you need to call this method from a user interface, you must call this method from a non-UI thread.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Executing  
Sequences Using API\DotNet\Executing Sequences Using  
API.seq
```

Highlighted Features

- TestStand API

Major API

- `Engine.GetSequenceFileEx`
- `Engine.NewExecution`
- `Engine.ReleaseSequenceFileEx`

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequences and steps in the example.

1. Open **Executing Sequences Using API.seq**.
2. On the Sequences pane, select the **MainSequence**. The `MainSequence` contains four Sequence Call steps that correspond to the different approaches demonstrated in this example.
3. On the Sequences pane, select the **Execute With No Process Model** sequence. This sequence uses the `Engine.GetSequenceFileEx` and `Engine.NewExecution` methods from the TestStand API to open and execute a sequence file. Because this type of execution does not use a process model, it is similar to selecting **Execute»Run MainSequence** in the target sequence file.
4. After execution completes, the `Engine.ReleaseSequenceFileEx` method releases the target sequence file. This is a necessary step for any sequence file loaded with the `Engine.GetSequenceFile` method.
5. On the Sequences pane, select the **Execute With Process Model** sequence. This sequence loads a process model with the `Engine.GetSequenceFileEx` method and uses the `Engine.NewExecution` method to execute a client sequence file using one of the entry points from the process model.
6. On the Sequences pane, select the **Execute With Sequence Parameters** sequence. This sequence passes parameters to the target sequence by using the `Engine.NewPropertyObject`, `PropertyObject.SetValNumber`, and `PropertyObject.SetValString` methods to create a container to store the parameters. The parameters must be declared in the same order they appear in the target sequence. Observe that the new `PropertyObject` container is passed to the `Engine.NewExecution` method as one of its parameters.
7. Open **Target Sequence File.seq**.
8. On the Sequences pane, select the **MainSequence**. This sequence includes a Message Popup step to indicate to the user that the sequence has executed.
9. On the Sequences pane, select the **Subsequence**. Observe that this sequence includes two parameters, which are displayed in the Message Popup step.

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to run the sequence.
2. As each section of code executes, the Message Popup from the target sequence appears. Click **OK** on these dialog boxes to continue execution.
3. When the `Execute With Sequence Parameters` sequence executes, enter a number and string in the Message Popups that appear. Notice that these values are displayed in the final Message Popup.

Related concepts:

- [TestStand Directory Structure](#)

Executing Sequences Using API - TestStand Expressions

Purpose

This example demonstrates various methods of calling a sequence using the TestStand API.



Note If you are developing a TestStand user interface which must handle TestStand UI messages, do not use the `WaitForEndEx()` method, since this method does not process the UI messages while it waits for the execution to complete. If you need to call this method from a user interface, you must call this method from a non-UI thread.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Executing  
Sequences Using API\TestStand Expressions\Executing  
Sequences Using API.seq
```

Highlighted Features

- TestStand API

Major API

- `Engine.GetSequenceFileEx`
- `Engine.NewExecution`
- `Engine.ReleaseSequenceFileEx`

Prerequisites

None

How to Use This Example

Complete the following steps to review the sequences and steps in the example.

1. Open **Executing Sequences Using API.seq**.
2. On the Sequences pane, select the **MainSequence**. The `MainSequence` contains four Sequence Call steps that correspond to the different approaches demonstrated in this example.
3. On the Sequences pane, select the **Execute With No Process Model** sequence. This sequence uses the `Engine.GetSequenceFileEx` and `Engine.NewExecution` methods from the TestStand API to open and execute a sequence file. Because this type of execution does not use a process model, it is similar to selecting **Execute»Run MainSequence** in the target sequence file.
4. After execution completes, the `Engine.ReleaseSequenceFileEx` method releases the target sequence file. This is a necessary step for any sequence file loaded with the `Engine.GetSequenceFile` method.
5. On the Sequences pane, select the **Execute With Process Model** sequence. This sequence loads a process model with the `Engine.GetSequenceFileEx` method and uses the `Engine.NewExecution` method to execute a client sequence file using one of the entry points from the process model.
6. On the Sequences pane, select the **Execute With Sequence Parameters** sequence. This sequence passes parameters to the target sequence by using the `Engine.NewPropertyObject`, `PropertyObject.SetValNumber`, and `PropertyObject.SetValString` methods to create a container to store the parameters. The parameters must be declared in the same order they appear in the target sequence. Observe that the new `PropertyObject` container is passed to the `Engine.NewExecution` method as one of its parameters.

7. Open **Target Sequence File.seq**.
8. On the Sequences pane, select the **MainSequence**. This sequence includes a Message Popup step to indicate to the user that the sequence has executed.
9. On the Sequences pane, select the **Subsequence**. Observe that this sequence includes two parameters, which are displayed in the Message Popup step.

Complete the following steps to run the example:

1. Select **Execute»Single Pass** to run the sequence.
2. As each section of code executes, the Message Popup from the target sequence appears. Click **OK** on these dialog boxes to continue execution.
3. When the `Execute With Sequence Parameters` sequence executes, enter a number and string in the Message Popups that appear. Notice that these values are displayed in the final Message Popup.

Related concepts:

- [TestStand Directory Structure](#)

Type Casting API Classes

The `<TestStand Public>\Examples\TestStand API\Type Casting API Classes` directory contains the following examples.

Related concepts:

- [TestStand Directory Structure](#)

Type Casting API Classes - LabWindows/CVI

Purpose

This example demonstrates how to type cast TestStand objects in order to access specific API properties and methods.

Example File Location

`<TestStand Public>\Examples\TestStand API\Type Casting API`

Classes\CVI\Type Casting API Classes.seq

Highlighted Features

- Built-In Properties
- Dynamic Properties
- RunState Property
- TestStand API

Major API

- PropertyObject.GetPropertyObject
- SequenceFile.AsPropertyObjectFile
- SequenceFile.AsPropertyObject

Prerequisites

None

How to Use This Example

Complete the following steps to review the code in this example:

1. Open the **Cast to More Generic Class** sequence from the Sequences pane. Select the **Cast to More Generic Class** step, then click **Edit Code** in the Module tab of the Step Settings pane to open the code module. This step casts a sequence file object to the more generic propertyObject and PropertyObjectFile classes to access their properties and methods.
2. Open the **Cast to More Specific Class** sequence from the Sequences pane. Select the **Cast to More Specific Class** step, then click **Edit Code** in the Module tab of the Step Settings pane to open the code module. This step casts a PropertyObject to more specific classes to access their properties and methods.

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. As the sequence executes, read the message popups for information about the kinds of type casting available in the TestStand API.

Related concepts:

- [TestStand Directory Structure](#)
- [Accessing Built-in Properties](#)
- [Accessing Dynamic Properties](#)
- [RunState Subproperties](#)

Type Casting API Classes - LabVIEW

Purpose

This example demonstrates how to type cast TestStand objects in order to access specific API properties and methods.

Example File Location

```
<TestStand Public>\Examples\\TestStand API\Type Casting API  
Classes\LabVIEW\Type Casting API Classes.seq
```

Highlighted Features

- Built-In Properties
- Dynamic Properties
- RunState Property
- TestStand API

Major API

- PropertyObject.GetPropertyObject
- SequenceFile.AsPropertyObjectFile
- SequenceFile.AsPropertyObject

Prerequisites

None

How to Use This Example

Complete the following steps to review the code in this example:

1. Open the **Cast to More Generic Class** sequence from the Sequences pane. Select the **Cast to More Generic Class** step, then click **Edit VI** in the Module tab of the Step Settings pane to open the code module. This step casts a sequence file object to the more generic `PropertyObject` and `PropertyObjectFile` classes to access their properties and methods.
2. Open the **Cast to More Specific Class** sequence from the Sequences pane. Select the **Cast to More Specific Class** step, then click **Edit VI** in the Module tab of the Step Settings pane to open the code module. This step casts a `PropertyObject` to more specific classes to access their properties and methods.

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. As the sequence executes, read the message popups for information about the kinds of type casting available in the TestStand API.

Related concepts:

- [TestStand Directory Structure](#)
- [Accessing Built-in Properties](#)
- [Accessing Dynamic Properties](#)
- [RunState Subproperties](#)

Type Casting API Classes - .NET

Purpose

This example demonstrates how to type cast TestStand objects in order to access specific API properties and methods.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Type Casting API  
Classes\dotNET\Type Casting API Classes.seq
```

Highlighted Features

- Built-In Properties
- Dynamic Properties
- RunState Property
- TestStand API

Major API

- PropertyObject.GetPropertyObject
- SequenceFile.AsPropertyObjectFile
- SequenceFile.AsPropertyObject

Prerequisites

None

How to Use This Example

Complete the following steps to review the code in this example:

1. Open the **Cast to More Generic Class** sequence from the Sequences pane. Select the **Cast to More Generic Class** step, then click **Edit in Visual Studio** in the Module tab of the Step Settings pane to open the code module. This step casts a sequence file object to the more generic propertyObject and PropertyObjectFile classes to access their properties and methods.
2. Open the **Cast to More Specific Class** sequence from the Sequences pane. Select the **Cast to More Specific Class** step, then click **Edit in Visual Studio** in the Module tab of the Step Settings pane to open the code module. This step casts a PropertyObject to more specific classes to access their properties and methods.

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. As the sequence executes, read the message popups for information about the kinds of type casting available in the TestStand API.

Related concepts:

- [TestStand Directory Structure](#)
- [Accessing Built-in Properties](#)
- [Accessing Dynamic Properties](#)
- [RunState Subproperties](#)

Type Casting API Classes - TestStand Expressions

Purpose

This example demonstrates how to type cast TestStand objects in order to access specific API properties and methods.

Example File Location

```
<TestStand Public>\Examples\TestStand API\Type Casting API  
Classes\TestStand Expressions\Type Casting API Classes.seq
```

Highlighted Features

- Built-In Properties
- Dynamic Properties
- RunState Property
- TestStand API

Major API

- PropertyObject.GetPropertyObject
- SequenceFile.AsPropertyObjectFile
- SequenceFile.AsPropertyObject

Prerequisites

None

How to Use This Example

Complete the following steps to review the code in this example:

1. Open the **Cast an Object Reference** sequence from the Sequences pane. Review the statement steps in this sequence. These steps cast a generic object reference to a specific class to access the properties of the specific class.
2. Open the **Cast to More Generic Class** sequence from the Sequences pane. Review the statement steps in this sequence. These steps cast a sequence file object to the more generic `propertyObject` and `PropertyObjectFile` classes to access their properties and methods.
3. Open the **Cast to More Specific Class** sequence from the Sequences pane. Review the statement steps in this sequence. These steps cast a `PropertyObject` to more specific classes to access their properties and methods.

Complete the following steps to run the example:

1. Select **Execute»Run MainSequence** to execute the sequence.
2. As the sequence executes, read the message popups for information about the kinds of type casting available in the TestStand API.

Related concepts:

- [TestStand Directory Structure](#)
- [Accessing Built-in Properties](#)
- [Accessing Dynamic Properties](#)
- [RunState Subproperties](#)

Examples for TestStand Debugging Features

The TestStand Debugging Features examples highlight TestStand features for troubleshooting your test sequences. This directory includes the following example programs.

Benchmarks

Purpose

This example performs a benchmark on each of the TestStand adapters.

Example File Location

```
<TestStand Public>\Examples\TestStand Debugging Features\  
Benchmarks\Benchmarks.seq
```

Highlighted Features

- Module adapters
- Performance

Major API

None

Prerequisites

You must log into TestStand as a user with administrator privileges. To use the LabVIEW benchmark portion of the example, you must have the LabVIEW development system or LabVIEW Run-Time Engine (RTE) installed and you must configure the LabVIEW Adapter to use the LabVIEW development system or LabVIEW RTE.

How to Use This Example

This example runs multiple sets of steps. Each set uses a different adapter and configuration. As the sequence executes, it records the total time required to execute each set of steps. The sequence generates a custom report to display the average execution time for a single step in each set of steps.

You can adjust the following options to improve performance:

- Disable the **Enable Tracing** option on the Execution tab of the Station Options dialog box.
- Enable the **Disable Result Recording for All Sequences** option on the Execution tab of the Station Options dialog box.
- To improve the performance of the ActiveX/COM Adapter, disable the **Use Late Binding** option in the ActiveX/COM Adapter Configuration dialog box.
- To improve the performance of the LabVIEW Adapter, enable the **Reserve Loaded VIs for Execution** option in the LabVIEW Adapter Configuration dialog box.

Complete the following steps to use this example.

1. Select **Execute»Single Pass** to run the sequence.
2. Select **Continue**. If you do not have the LabVIEW development system or LabVIEW RTE installed, select the **Continue - skip LabVIEW** option.
3. When execution completes, review the report on the Report pane.

Related concepts:

- [Module Adapters](#)
- [TestStand Directory Structure](#)
- [Privileges](#)
- [Programming with the TestStand API in LabVIEW](#)

Custom Analyzer Rules

Purpose

This example demonstrates how to implement custom TestStand Sequence Analyzer rules and analysis modules in LabVIEW, LabWindows/CVI, Microsoft Visual C#, and Microsoft Visual C++. The example directory contains the following example custom rules:

- **CheckNumberOfLocals**—This rule verifies that the number of locals in each analyzed sequence does not exceed a maximum number. The maximum is a configurable rule setting that users edit with a standard built-in dialog box. This example demonstrates how to use rule settings to configure rules and how to use analysis transitions to perform bookkeeping tasks.
- **CheckStepNameLength**—This rule verifies that the analyzed step names do not exceed a maximum length. The maximum length is a configurable rule setting that users edit using a custom dialog box. This example demonstrates how to use a rule configuration module to configure rules using a custom dialog box and how to store and retrieve rule configuration data.

Each example rule contains source code for analysis modules and rule configuration modules written in LabVIEW, LabWindows/CVI, Microsoft Visual C#, and Microsoft Visual C++.

Example File Location

This example does not use a separate sequence file. Create a new sequence file and follow the steps in to import the files located in the <TestStand Public>\Examples\TestStand Debugging Features directory.

Highlighted Features

- Analysis module
- Analysis transitions
- Rule configuration module
- Rule settings

Major API

- AnalysisContext
- RuleConfiguration
- RuleConfigurationContext

Prerequisites

You do not need any additional software to run the examples. You must have a supported version of LabVIEW, LabWindows/CVI, or Microsoft Visual Studio installed to modify and rebuild the modules.

How to Use This Example

This example implements several rules and analysis modules using LabVIEW, LabWindows/CVI, Visual C#, and Visual C++. You must import the corresponding rules files in the Configure Sequence Analyzer Available Rules dialog box to add the example rules to the sequence analyzer.

After you add the rule to the sequence analyzer, the analyzer adds the rule to all opened analyzer projects. The sequence analyzer uses the analysis module the next time you analyze a project.

Related concepts:

- [TestStand Directory Structure](#)

Generating Output Messages

Purpose

This example demonstrates how to generate output messages using the `OutputMessage` expression function and the `Engine.NewOutputMessage` and `OutputMessage.Post` methods of the TestStand API. The Output pane of the TestStand Sequence Editor displays the messages.

Example File Location

```
<TestStand Public>\Examples\TestStand Debugging Features\  
Generating Output Messages\Generating Output Messages.seq
```

Highlighted Features

- Action step
- ActiveX/COM Adapter
- Sequence Call step
- Statement step

Major API

- `Engine.NewOutputMessage`
- `OutputMessage.Post`

Prerequisites

(LabVIEW) You must have the LabVIEW development system installed and you must configure the LabVIEW Adapter to use the LabVIEW development system.

(LabWindows/CVI) You must have the LabWindows/CVI Run-Time Engine installed and you must configure the LabWindows/CVI Adapter to execute steps in-process. If you want to use the **Execute Steps in an External Instance of CVI** option, you must have the LabWindows/CVI development environment installed.

How to Use This Example

None

LabVIEW Action Step

Open **OutputMessageExample.vi**, located in the <TestStand Public>\Examples\TestStand Debugging Features\Generating Output Messages directory, and switch to the block diagram, which describes the following functionality for the VI:

- The invoke node labeled IEngine invokes the `Engine.NewOutputMessage` method.
- Parameters that enter the VI determine the sequence for which the message is intended and the content, category, and severity of the message.
- The subsequent property nodes specify color and icon name.
- The final invoke node invokes the `OutputMessage.Post` method to display the output message on the Output pane in the sequence editor

LabWindows/CVI Action Step

The LabWindows/CVI Action step calls a LabWindows/CVI DLL named `GenerateOutputMessages.dll`, located in the <TestStand Public>\Examples\TestStand Debugging Features\Generating Output Messages directory.

Select the **Generate Output Messages using a CVI Module** step and click the LabWindows/CVI Module tab on the Step Settings pane, which describes the following functionality for the LabWindows/CVI step:

- The Module control specifies `GenerateOutputMessages.dll`.
- The Function control specifies the `GenerateOutputMessages` function.
- The Parameters table lists the parameters the DLL accepts. The **SeqContextCVI** parameter accepts the sequence context, and the other parameters handle errors or log information for inclusion in the report.

Statement Step

The Statement step is the simplest way to generate output messages. Select the **Generate Output Message using a Statement step** step and click the Expression edit tab on the Step Settings pane, which describes the following functionality for the Statement step:

- The expression specifies a simple comparison to determine if the `RunState.LoopIndex` value is a multiple of 10. If true, the modulus 10 operation evaluates to 0, and TestStand calls the `OutputMessage` function.
- The parameters of the `OutputMessage` function accept a message and a severity, and specify the font color and icon to display with the message.

ActiveX/COM Steps

The final step in the `MainSequence` is a Sequence Call step that calls the Using ActiveX Adapter steps sequence. Select the **Using ActiveX Adapter steps** on the Sequences pane to display the steps in the Using ActiveX Adapter steps sequence, which describes the following functionality:

- A For loop contains all the steps in the sequence. The If step specifies that the steps execute during only every tenth iteration through the loop.
- The New Output Message step creates the output message by calling the `Engine.NewOutputMessage` method. This step also specifies the message text, category, and severity and obtains a reference to the sequence context.
- The Set Text Color step uses the Set Property option on the `TextColor` property to change the message font color to cyan.
- The Set Icon Name step specifies the icon to display next to the message.
- The Post Output Message step invokes the `OutputMessage.Post` method to display the output message on the Output pane. Simply creating the message using the `Engine.NewOutputMessage` method is insufficient.

Related concepts:

- [TestStand Directory Structure](#)
- [Sequence Context](#)

TestStand API Overview

Learn basic information about the TestStand ActiveX Automation server, API concepts, and writing applications with TestStand.

Using TestStand API Objects

In the TestStand API, many objects reference other types of objects. You can obtain an object of one class from an object of another class by calling a method or accessing a property. For example, you can obtain the Engine, Thread, Execution, and Report objects from the SequenceContext object. In another example, a UIMessage object references an `Execution` object, and an `Execution` object references a `SequenceFile` object.

Related concepts:

- [Getting Started with TestStand](#)

TestStand ActiveX API Overview

You can use the TestStand ActiveX Automation server API to access the internal data of TestStand sequences and steps or to create and control sequence executions.

ActiveX Automation Server Concepts

An ActiveX Automation server is any executable code that other applications can access. ActiveX uses a client/server relationship in which the client requests objects from the server and sends commands to the server to perform actions on the objects.

Clients and servers can run on a single computer or on separate computers connected by a network. An ActiveX server is local when it runs on the same computer as its client and remote when it runs on a different computer.

On a single computer, an ActiveX server runs in-process or out-of-process with respect to the client. An in-process ActiveX server is a DLL that runs in the same process space

as the client. An out-of-process ActiveX server is an executable and therefore does not run in the same process space as the client.



Note Using an out-of-process server can add significant performance overhead to the client application.

Related concepts:

- [Objects, Classes, Methods, Properties, and Controls](#)
- [Object References](#)
- [Adding and Releasing References](#)

Objects, Classes, Methods, Properties, and Controls

The interface for an ActiveX Automation server includes the following components:

- **Object**—A programmable software component that represents an instance of a class. You interact with the object using the methods, properties, and events the class defines.
- **Class**—Similar to a data type definition, except that a class applies to objects rather than variables. A class defines a list of methods and properties you can use with respect to the objects you create as instances of the class. For some classes, you can create objects directly. For other classes, you can obtain objects only by calling a method or property on an existing object of another class.
- **Method**—Performs an operation or function on an object.
- **Property**—A value that stores and maintains a setting or attribute of an object.
- **Control**—A visible or invisible ActiveX component you use on user interface windows.

Object References

All ActiveX objects maintain reference counts. Each object keeps track of the number of other objects that reference it so the object can determine when to free the resources it uses. Typically, you do not have to take any specific actions to maintain a reference count. However, you must help maintain the reference count if you call a method or property that returns an object or if you store a reference to an object in a global variable for use after the test module completes.

Related concepts:

- [Adding and Releasing References](#)

Adding and Releasing References

Whenever you create an ActiveX object directly or receive an ActiveX object as a return value from a method or property, you have a reference to the object. You must release this reference when you no longer require the object.

Normally, when you receive an object as a parameter to a function or LabVIEW VI, you do not need to add or release a reference to the object because the caller of the function already holds a reference to the object. If, however, the function or VI stores the object for later use after the function or VI returns, you must add a reference to the object in the function or VI.

In general, maintaining a reference to an object guarantees that the object exists for at least as long as you hold the reference. The object frees its resources only when you release the final reference to the object.

Refer to the following examples for more information about adding and releasing references in specific development environments:

- LabVIEW
- LabWindows/CVI
- Microsoft Visual Basic .NET
- C#
- Microsoft Visual C++/#import

Related concepts:

- [Releasing ActiveX References in LabVIEW](#)
- [Adding and Releasing References in LabWindows/CVI](#)
- [Adding and Releasing References in Microsoft Visual Basic .NET](#)
- [Adding and Releasing References in C#](#)
- [Adding and Releasing References in Microsoft Visual C++/#import](#)

Releasing ActiveX References in LabVIEW

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



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

Adding and Releasing References in LabWindows/CVI

LabWindows/CVI automatically maintains an object reference for each handle you obtain for an object. When you assign the handle to another variable, LabWindows/CVI does not add a reference to the object. Use the `CA_DuplicateObjHandle` function in the LabWindows/CVI ActiveX Automation Library to obtain a new handle to an existing object, which adds a reference to the object.

LabWindows/CVI automatically releases the object reference for each handle you obtain when you call the `CA_DiscardObjHandle` function from the LabWindows/CVI ActiveX Automation Library.

The following example shows how to obtain a handle to the TestStand Engine from the SequenceContext object, how to call a method on the engine to acquire a version string, and how to release the handle to the engine and the string.

```
int GetEngineVersion(CAObjHandle seqContextCVI)

{

int error = 0;

ErrMsg errMsg = "";
```

```

ERRORINFO errorInfo;

CAObjHandle engine = 0;

char *versionString = NULL;

tsErrChk(TS_SeqContextGetEngine(seqContextCVI, &errorInfo,
&engine));

tsErrChk(TS_EngineGetVersionString (engine, &errorInfo,
&versionString));

Error:

// Free Resources

if (engine)

CA_DiscardObjHandle(engine);

if (versionString)

CA_FreeMemory(versionString);

return error;

}

```



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

While many of the functions specified in the `tsapicvi.fp` library are simple wrappers to API methods that require no storage of information, there are several functions, especially those containing `Get` or `New`, where TestStand is actively allocating new memory to hold the information. In any instance where you are using a function of this type, you must release the allocated memory at the end of the code

using calls to `CA_FreeMemory`, `CA_DiscardObjHandle`, or similar functions.

When you are concerned about a function returning a piece of data that must be manually released, refer to the LabWindows/CVI Help or to the NI TestStand Help for that function. Both of these resources explicitly state if the function is allocating memory and often contain additional code fragments explaining how to use the function.

The following are examples of functions that allocate memory:

- `TS_PropertyGetValString()`
- `TS_PropertyGetValIDispatch()`
- `TS_PropertyGetPropertyObject()`
- `TS_NewEngine()`
- `TS_SeqFileNewEditContext()`
- `TS_EngineNewSequence()`

The following example uses one of the previous functions and then releases the memory:

```
char *stringVal = NULL;

TS_PropertyGetValString (propObj, &errorInfo,
"Step.Limits.String", 0, &stringVal);

...

CA_FreeMemory (stringVal);
```

Adding and Releasing References in Microsoft Visual Basic .NET

Microsoft Visual Basic .NET automatically adds object references when you assign the object to another variable, as shown in the following code:

```
Set ReferenceObj = OtherReferenceObj
```

Visual Basic .NET also automatically releases object references when a variable or list that contains a reference to an object goes out of scope. Local variable scope ends when the function returns. You can initiate the release of a reference to an object by setting the variable to the value `Nothing`, as shown in the following code:

```
Set ReferenceObj = Nothing
```

Adding and Releasing References in C#

C# automatically adds object references when you assign the object to another variable, as shown in the following code:

```
referenceObj = otherReferenceObj;
```

C# also automatically releases object references when a variable or list that contains a reference to an object goes out of scope. Local variable scope ends when the function returns. You can initiate the release of a reference to an object by setting the variable to the value `null`, as shown in the following code:

```
referenceObj = null;
```

Adding and Releasing References in Microsoft Visual C++/#import

The pointer classes the `#import` directive generates automatically add object references when you construct a pointer, as shown in the following code:

```
TS::PropertyObjectPtr newObjectPtr(objectPtr);
```

The assignment operator of the pointer classes first releases the reference to the object it currently holds and then adds a reference to the object being assigned to it. For example, the following assignment statement first releases the reference to the object to which `newObjectPtr` refers and then adds a reference to the object to which `objectPtr` refers:

```
newObjectPtr = objectPtr;
```

The destructor for the pointer class releases the object reference. Assigning the value `NULL` to a pointer variable also releases the object reference, as shown in the following example:

```
objectPtr = NULL; // releases the reference
```

TestStand API Concepts

The Core API Classes, Properties, and Methods topic includes a complete listing of the classes in the core TestStand API. The Adapter API Classes, Properties, and Methods topic includes a listing of adapter classes.

Accessing Built-in Properties

TestStand defines a number of built-in properties that always exist for objects, such as the `Sequence.Name` property or the `SequenceContext.Sequence` property. Almost every TestStand object defines built-in, static API properties you can access in the programming language you use to create code modules.

Refer to the following examples for more information about accessing built-in properties in specific development environments:

- LabVIEW
- LabWindows/CVI
- Microsoft Visual Basic .NET
- C#
- Microsoft Visual C++/#import

Related concepts:

- [Accessing Built-in Properties in LabVIEW](#)
- [Accessing Built-in Properties in LabWindows/CVI](#)
- [Accessing Built-in Properties in Microsoft Visual Basic .NET](#)
- [Accessing Built-in Properties in C#](#)
- [Accessing Built-in Properties in Microsoft Visual C++/#import](#)

Accessing Dynamic Properties

You can define custom step properties, sequence local variables, sequence file global variables, and station global variables that are dynamic with respect to the TestStand API. The TestStand API provides the `PropertyObject` class so you can access dynamic properties and variables in the programming language you use to create code modules. Instead of using defined constants, use lookup strings to identify specific properties by name.

Refer to the following examples for more information about accessing dynamic properties in specific development environments:

- LabVIEW
- LabWindows/CVI
- Microsoft Visual Basic .NET
- C#
- Microsoft Visual C++/#import

Related concepts:

- [Lookup Strings](#)
- [Accessing Dynamic Properties in LabVIEW](#)
- [Accessing Dynamic Properties in LabWindows/CVI](#)
- [Accessing Dynamic Properties in Microsoft Visual Basic .NET](#)
- [Accessing Dynamic Properties in C#](#)
- [Accessing Dynamic Properties in Microsoft Visual C++/#import](#)

Acquiring a Derived Class from the PropertyObject Class

In some cases, you can obtain a reference to a TestStand object using the `PropertyObject` class methods and then access one of the static properties of the object. For example, you might want to obtain the run mode for the third step of the Main step group of the currently executing sequence. For methods on the `PropertyObject` class that can return objects derived from the `PropertyObject` class, you must acquire the derived interface for the object to access the built-in properties and methods of the derived class.

Refer to the following examples for more information about acquiring a derived class from the `PropertyObject` class in specific development environments:

- LabVIEW
- Microsoft Visual Basic .NET
- C#
- Microsoft Visual C++/#import

Related concepts:

- [Acquiring a Derived Class from the PropertyObject Class in LabVIEW](#)
- [Acquiring a Derived Class from the PropertyObject Class in Microsoft Visual Basic .NET](#)
- [Acquiring a Derived Class from the PropertyObject Class in C#](#)
- [Acquiring a Derived Class from the PropertyObject Class in Microsoft Visual C++/#import](#)

Object Relationships

The `Engine` class is the only class in the core TestStand API for which you can create an object directly. The `Engine` class defines methods for creating objects for the other classes. In some cases, you can obtain objects for one class from another class. For example, you can obtain a `Step` object from the `Sequence` object that contains the step.

API Inheritance

Almost all TestStand API classes inherit from the `PropertyObject` class, which means you can use `PropertyObject` methods and properties on objects of other classes. For example, you can use the `PropertyObject.Name` property to obtain the name of a `SequenceFile` object, `Sequence` object, or `Step` object. You can also create subproperties using the `PropertyObject` methods. You can programmatically create a new subproperty of a step using the `PropertyObject` methods `PropertyObject.SetValNumber`, `PropertyObject.SetValBoolean`, or `PropertyObject.SetValString` with the `PropOption_InsertIfMissing` option.

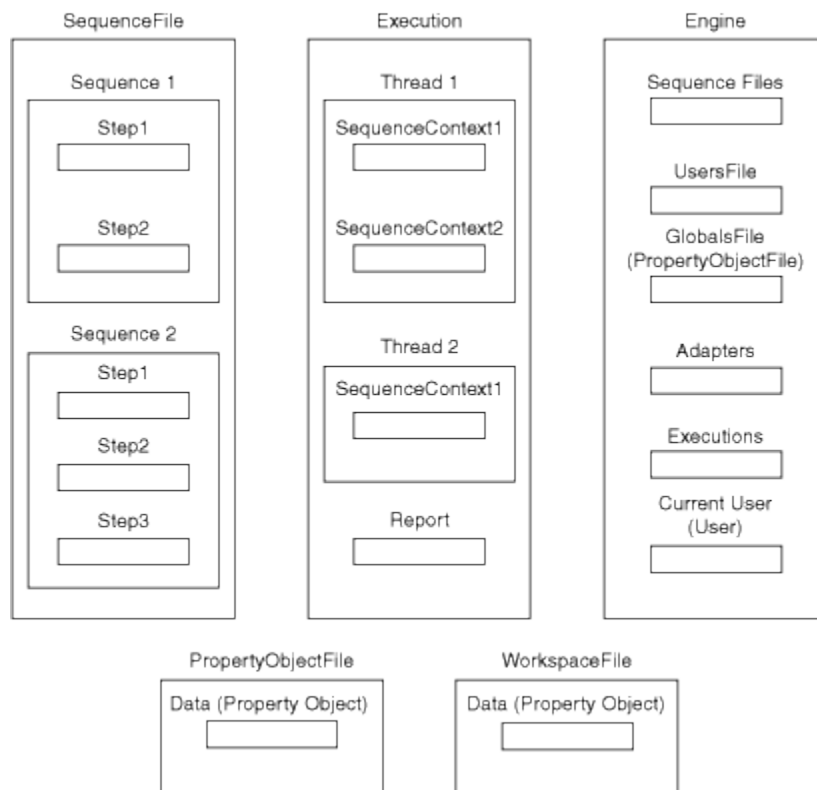


Note In most development environments, you must use the

`AsPropertyObject` method to obtain a `PropertyObject` interface for the original object. In LabWindows/CVI, you can use the `PropertyObject` methods and property functions without first obtaining the `PropertyObject` interface by using the `AsPropertyObject` method.

API Containment

In the TestStand API, many objects contain other types of objects. For example, a `SequenceFile` object contains `Sequence` objects, and `Sequence` objects contain `Step` objects. The following figure shows the containment relationship between objects in the TestStand API.



Property Paths

A property path is a chain of period-delimited subproperties that selects a particular subproperty of an object. A property path typically specifies the location of a property or variable in the active sequence context. For example, the active sequence context contains the Station Globals object, which contains a `TS` property that holds the current `User` object. To refer to the current `User` object, use the

`StationGlobals.TS.CurrentUser` property path.

You can use the sequence context and property paths to access variables and step properties in expressions and through calls to the TestStand API from code modules.

You can specify a complete path, regardless of the length or complexity of the path, when all of the elements in the path are dynamic properties. For example, to specify the error code in the third step of the Main step group of the currently executing sequence, use the following path:

```
"RunState.Sequence.Main[2].Result.Error.Code"
```

Related concepts:

- [Expressions](#)
- [Sequence Context](#)

Finding a Property Path

Because the existence or content of some properties might vary at run time, you can browse the sequence context during the execution of the sequence to locate a specific property path.

Complete the following steps to locate a property path.

1. Set a breakpoint on the step in which you intend to use the property path and then execute the sequence.
2. When TestStand reaches the breakpoint, click the Variables pane in the Execution window and browse to the property.
3. To copy the path to the property to the system clipboard, select **Copy** from the context menu of the property. You can then paste the path into a text field in TestStand, such as an expression, or into the source code for a code module. If you paste into the Variables pane, TestStand inserts a copy of the object.

When searching for a property path, you must consider the method by which you execute the sequence. For example, one common property path is that of the unit under test (UUT) serial number. The local variable that contains the serial number exists within the Test UUTs Execution entry point sequence of the process model. You

can only browse to this property while executing the sequence using the Test UUTs Execution entry point. The property path of the UUT serial number is `RunState.Root.Locals.UUT.SerialNumber`. In this case, the `RunState.Root` property contains the sequence context of the Test UUTs Execution entry point.

Related concepts:

- [Sequence Context](#)

Lookup Strings

To specify the dynamic property on which a method operates, pass a string that defines a complete path from the object on which you call the method to the specific property you want to access. This string is called a lookup string. Use the following techniques to specify different objects:

- To specify the object itself, pass an empty string ("").
- To specify a subproperty, pass the name of the subproperty.
- To specify a subproperty of a subproperty, pass a string that contains both names separated by a period (.).

For example, you can specify the error code property in a step object using the following lookup string:

```
"Result.Error.Code"
```

TestStand expressions require `PropertyObject` names to be strings that contain only letters, numbers, and underscores. `PropertyObject` names cannot contain spaces, start with a number, or be empty. For example, `MyVariableName_2000` is a valid `PropertyObject` name. Names of elements in a `PropertyObject` array, such as sequence and step names, can contain any character. Instead of using lookup strings that contain array element names that might not be unique, such as step names, with methods that have a **lookupString** parameter, use a string that specifies the array index instead.



Note TestStand does not generate an error if you programmatically set an

invalid name for a `PropertyObject`.

You can specify an attribute or type attribute using a lookup string. Use `Attributes` in a lookup string to specify the attributes of the property object and use `TypeAttributes` to specify the type attributes of the property object. For example, the following lookup string accesses an attribute of a local variable from a sequence:

```
"Locals.Foo.Attributes.MyAttributeNameSpace.Attribute1"
```



Note If a property object defines a subproperty named `Attributes` or `TypeAttributes`, the subproperty takes precedence over the attributes or type attributes in a lookup string.

When you specify a named array element using a lookup string, such as `myarray["myelementname"]`, use the standard C-style backslash escaping technique to escape special characters in the string. For example, for a single backslash in the actual name, use two backslashes in the escaped name. Use the `Utility.Escape` method to escape a string as needed.

Related concepts:

- [Accessing Dynamic Properties](#)

Thread Concurrency Models

In ActiveX, you can initialize a thread to use the single threaded apartment (STA) concurrency model or the multithreaded apartment (MTA) concurrency model. Before you use a thread with ActiveX, you must initialize the thread to one of these models. Typically, the software development environment you are using initializes the threads in the application for you.

When using the TestStand API to implement a user interface or other application program, you can use threads of either concurrency model.

When using ActiveX automation from within a code module, you do not have to initialize the thread concurrency model because the initial thread in an execution is

initialized as multithreaded and TestStand initializes new threads in an execution depending on the option you select in the Sequence Call Advanced Settings window.

When using ActiveX controls on a window or dialog box a code module creates, you must ensure that the thread is initialized as STA. If TestStand preloads a code module when an execution starts, the root execution, which is initialized as MTA, typically loads the code module. When a code module is loaded dynamically, the thread that first executes the step loads the code module.

You must complete the following steps to ensure that a code module is loaded by an STA thread.

- Execute the step in an STA thread. Use a Sequence Call step to create a STA thread by enabling the **Use Single-Threaded Apartment** option for the step in the Thread Settings dialog box.
- Select **Load Dynamically** for the **Load Option** option in the Step Properties dialog box.

For LabVIEW VIs that contain ActiveX controls, set the **Preferred Execution System** option for the VI to **user interface** to ensure that the VI executes within the user interface thread, which is typically an STA thread.



Note When you call a LabVIEW VI that creates ActiveX references, such as calling the ApplicationMgr.GetEngine method directly, you configure the VI to use the **same as caller** execution system, and you call the VI from a sequence running in a single-threaded apartment thread, LabVIEW might seem to hang with 100% processor use. Change the **Preferred Execution System** option in LabVIEW to a value different from **same as caller** to work around this issue.

Related concepts:

- [Launching an MFC Dialog with ActiveX Controls](#)

Programming with the TestStand API in Different Languages

You can use the TestStand API in the following programming languages.

Related concepts:

- [Accessing Properties Using API](#)

Programming with the TestStand API in LabVIEW

You can use the TestStand API or TestStand User Interface (UI) Controls from LabVIEW test and user interface VIs.

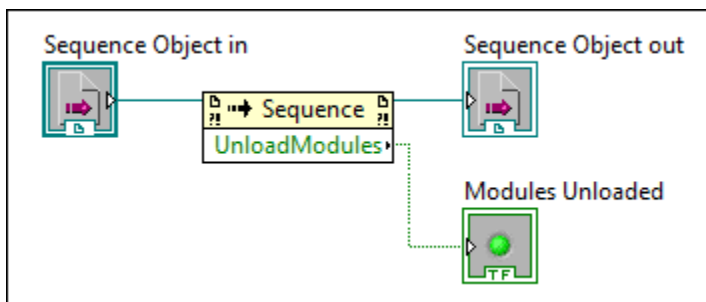
Refer to the LabVIEW documentation for more information about ActiveX concepts and how to use LabVIEW as an ActiveX client.

Related concepts:

- [TestStand UI Controls](#)
- [Programming with the TestStand API in LabVIEW](#)

Invoking Methods in LabVIEW

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

**Related concepts:**

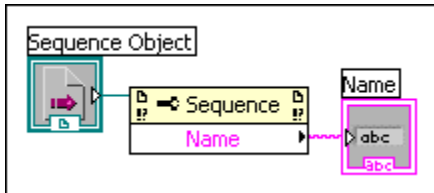
- [Setting the Preferred Execution System of LabVIEW VIs to Prevent Deadlock](#)

Accessing Built-in Properties in LabVIEW

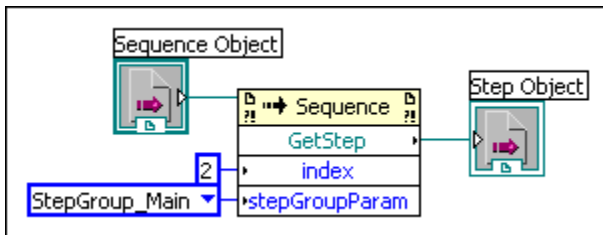
TestStand defines a number of built-in properties that are always present for objects,

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

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



The block diagram in the following figure shows how to obtain a reference to a step of a sequence that a Sequence object references.

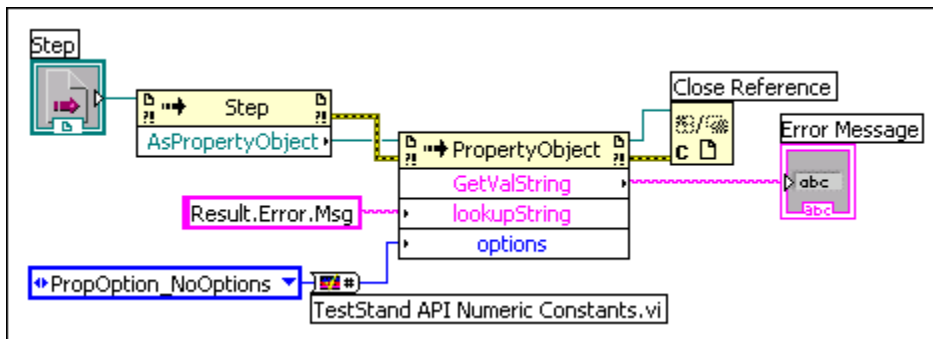


Accessing Dynamic Properties in LabVIEW

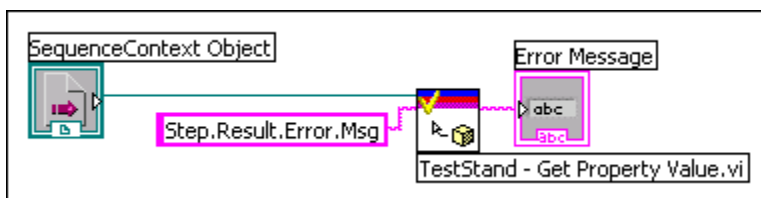
In TestStand, you can define custom step properties, sequence local variables, sequence file global variables, and station global variables. Because the TestStand API is independent of the variables and custom step properties you define, these variables and properties are dynamic with respect to the TestStand API. The TestStand API provides the `PropertyObject` class so you can access dynamic properties and variables from within code modules, where you use lookup strings to identify specific properties by name.

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

The block diagram in the following figure shows how to use the GetValString method on the PropertyObject interface of a Step object to obtain the error message value for the current step.



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



Programming with TestStand API Constants and Enumerations in LabVIEW

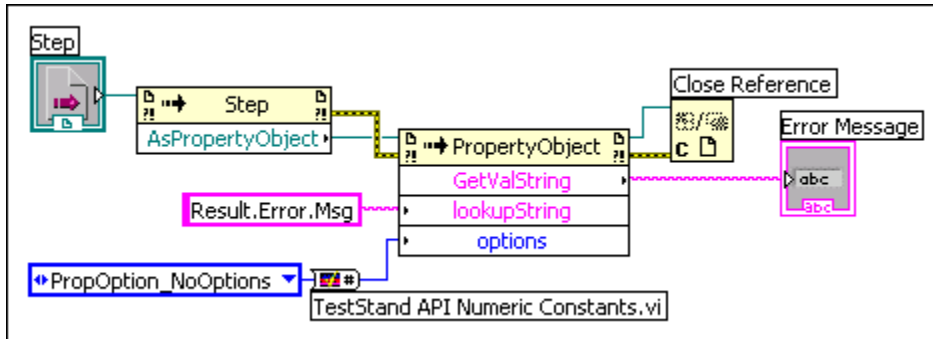
Some TestStand API methods require string and numeric constant input arguments. The acceptable values of these arguments are organized into groups that correspond to different properties and methods. For example, the `PropertyObject.SetValNumber` method includes an options input argument that accepts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

To facilitate programming with the TestStand API within VIs and to help you manage all the available string and numeric constants for the TestStand API properties and methods, TestStand provides the following enumerated constant VIs:

- **TestStand API String Constants VI**—Locate and select the string constant arguments you can use with TestStand API properties and methods.
- **TestStand API Numeric Constants VI**—Locate and select the various numeric constant arguments you can use with TestStand API properties and methods.

Use both of these VIs in conjunction with the constants associated with the TestStand API methods and properties.

The block diagram in the following figure shows how to use the TestStand API Numeric Constants VI to obtain the value of the PropOptions_NoOptions constant.



Combining Multiple Numeric Constants

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

Creating Enumeration Input Parameters

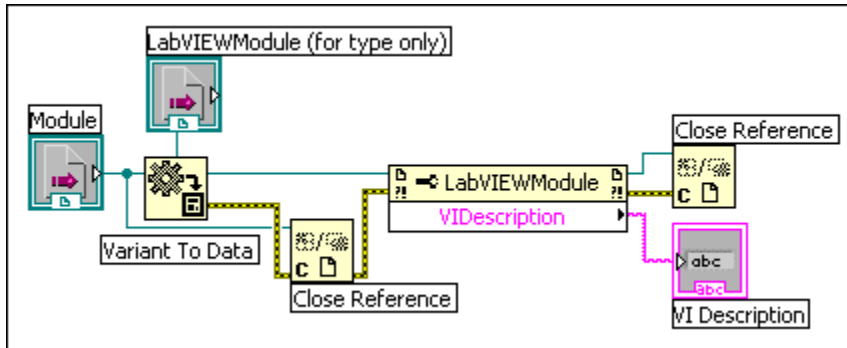
Some TestStand API methods require enumeration input arguments. For these methods, right-click the input parameter on the Invoke Node in LabVIEW, select **Create»Constant** from the context menu to create a LabVIEW ring constant, and select the value you want in the resulting constant.

Obtaining a Different Interface for TestStand Objects in LabVIEW

In some cases, you might need to obtain a different interface for a TestStand object than the current interface. In ActiveX/COM terminology, this action is called a QueryInterface.

For example, when you have a Module reference to a LabVIEWModule object and need to access the LabVIEWModule interface instead, perform a QueryInterface on the Module object to obtain the LabVIEWModule interface. In LabVIEW, use the Variant To Data function with the reference to accomplish this task.

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



Selecting an ActiveXServer Class

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

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

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

Acquiring a Derived Class from the PropertyObject Class in LabVIEW

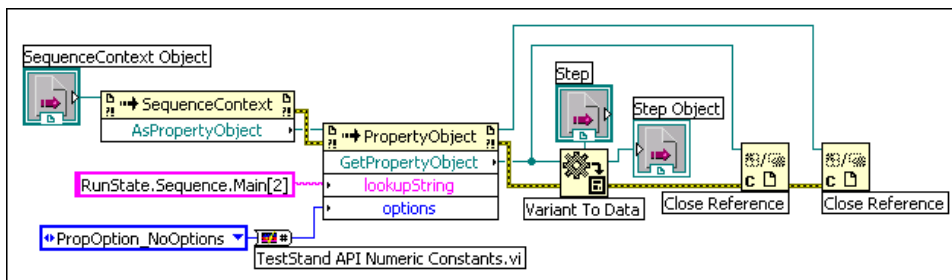
In some cases, you might need to use the `PropertyObject` class methods to obtain a reference to a TestStand object. You might then want to access one of the static properties of the TestStand object, such as the run mode for the third step in the Main step group of the currently executing sequence.

For methods in the `PropertyObject` class that can return objects derived from `PropertyObject`, you must acquire the derived interface for the object to access the built-in properties and methods of the derived class.

The block diagram in the following figure shows how to use a lookup string to obtain a reference to a `Step` object from a `SequenceContext` object.



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



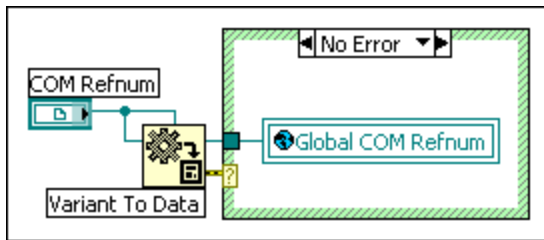
Related concepts:

- [Obtaining a Different Interface for TestStand Objects in LabVIEW](#)

Duplicating COM References in LabVIEW

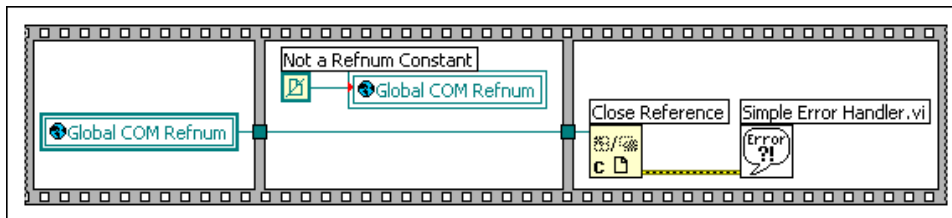
When TestStand passes an ActiveX reference to a LabVIEW code module, the reference automatically closes when the called VI completes executing and returns to TestStand. Wiring the reference to a global variable or to a shift register in another VI does not prevent the reference from closing. Consequently,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

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



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

When LabVIEW no longer needs the duplicate reference, read the global variable and wire the reference to the LabVIEW Close Reference function to close the reference. Wire the global variable to the LabVIEW Not A Refnum Constant to close the global variable, as shown in the following figure.



Setting the Preferred Execution System of LabVIEW VIs to Prevent Deadlock

When a VI calls methods of the TestStand API, you must correctly set the Preferred Execution System on the Execution page of the VI Properties dialog box for the VIs.

When you call synchronous methods that do not return until the LabVIEW server executes a VI on behalf of TestStand, the VI that calls these methods and the VI that TestStand attempts to run using the LabVIEW VI Server cannot run in the same LabVIEW execution system. When the VIs attempt to run in the same execution system, the execution of the synchronous TestStand method consumes the execution system in which the VI needs to run, causing deadlock. Since most TestStand API methods are synchronous, you should consider this when calling any TestStand API method from a LabVIEW user interface.

Because LabVIEW handles ActiveX communication through the user interface execution system, you cannot set either of the VIs in this scenario to run in the user interface execution system. For example, you can have a LabVIEW code module that

calls the `Engine.NewExecution` method followed by the `Execution.WaitForEndEx` method and a new execution that calls LabVIEW code modules. Deadlock can occur when either VI in this scenario uses **same as caller** or **user interface** as the preferred execution system.

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

Programming with the TestStand API in LabWindows/CVI

You can use the TestStand API or TestStand User Interface (UI) Controls from LabWindows/CVI code modules and user interface source code.

Refer to the ActiveX Library topic of the LabWindows/CVI Online Help for fundamental information about ActiveX concepts and how to access ActiveX servers from LabWindows/CVI.

Related concepts:

- [TestStand UI Controls](#)
- [Programming with the TestStand API in LabWindows/CVI](#)

Using ActiveX Drivers in LabWindows/CVI

LabWindows/CVI creates and accesses ActiveX objects using functions in a LabWindows/CVI-generated driver. This driver uses function panels to define C functions for all the methods and properties available for each object. For servers that define events, the driver contains functions for registering callback sequences for events.

The driver functions you use to invoke methods and properties have the following special naming conventions:

- Function names start with a prefix, such as `TS_`.
- Methods are followed by the class name and the method name.
- Properties are followed by `Get` or `Set` and the property name.
- In some cases, the class, method, and property names are abbreviated to keep the function name within the constraints of the `.fnp` file format.

The LabWindows/CVI ActiveX Automation Library uses the `CAObjHandle` data type for handles to ActiveX objects. The TestStand ActiveX drivers also follow this convention, so you can use the `CAObjHandle` data type for all handles to TestStand objects. However, one drawback of using the same data type for all TestStand objects is that the compiler cannot flag calls to methods in which you pass a handle for the wrong kind of object.

Objects can support more than one interface. For example, a `SequenceContext` object has a `SequenceContext` interface and a `PropertyObject` interface. When using handles in LabWindows/CVI to invoke methods or access properties of an object, you do not have to convert a specific reference for one interface to a specific reference for another interface. The ActiveX driver always queries the handle for the proper interface before invoking the method or accessing the property.

When you receive an object handle as the result of calling a method or obtaining the handle from a property, you must release the handle when you are finished with it.

Some TestStand ActiveX API methods include output parameters that return strings. You must use the `CA_FreeMemory` function in the LabWindows/CVI ActiveX Automation Library to free these strings when you are done with them.

Related concepts:

- [Adding and Releasing References in LabWindows/CVI](#)

Invoking Methods in LabWindows/CVI

TestStand objects have methods you invoke to perform an operation or function. In LabWindows/CVI, you invoke methods on TestStand objects using the functions the ActiveX driver for those objects defines.

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

```
int GetNumSteps(CAObjHandle sequence)

{

int error = 0;
```

```

ErrMsg errMsg = "";

ERRORINFO errorInfo;

CAObjHandle engine = 0;

long numSteps = 0;

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

Error:

return error;

}

```

The `errorInfo` variable is a structure the LabWindows/CVI ActiveX Automation Library defines to hold information about errors that can occur in the operation of the function. The `tsErrChk` macro determines whether the function return value or the `errorInfo` variable indicates an error occurred and continues execution at the `Error` label when `True`.



Note The functions, constants, and enumerations in the `tsapicvi.fpx` driver begin with the unique prefix `TS_`. This prefix is not included in the function, constant, and enumeration names in the NI TestStand Help.

Related concepts:

- [Accessing Built-in Properties in LabWindows/CVI](#)
- [Accessing Dynamic Properties in LabWindows/CVI](#)
- [ActiveX Automation Server Concepts](#)
- [Using ActiveX Drivers in LabWindows/CVI](#)
- [Adding and Releasing References in LabWindows/CVI](#)
- [Programming with TestStand API Constants and Enumerations in LabWindows/CVI](#)

Adding and Releasing References in LabWindows/CVI

LabWindows/CVI automatically maintains an object reference for each handle you obtain for an object. When you assign the handle to another variable, LabWindows/CVI does not add a reference to the object. Use the `CA_DuplicateObjHandle` function in the LabWindows/CVI ActiveX Automation Library to obtain a new handle to an existing object, which adds a reference to the object.

LabWindows/CVI automatically releases the object reference for each handle you obtain when you call the `CA_DiscardObjHandle` function from the LabWindows/CVI ActiveX Automation Library.

The following example shows how to obtain a handle to the TestStand Engine from the `SequenceContext` object, how to call a method on the engine to acquire a version string, and how to release the handle to the engine and the string.

```
int GetEngineVersion(CAObjHandle seqContextCVI)
{
    int error = 0;

    ErrMsg errMsg = "";

    ERRORINFO errorInfo;

    CAObjHandle engine = 0;

    char *versionString = NULL;

    tsErrChk(TS_SeqContextGetEngine(seqContextCVI, &errorInfo,
    &engine));

    tsErrChk(TS_EngineGetVersionString (engine, &errorInfo,
    &versionString));

    Error:
```

```
// Free Resources

if (engine)

CA_DiscardObjHandle(engine);

if (versionString)

CA_FreeMemory(versionString);

return error;

}
```



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

While many of the functions specified in the `tsapicvi.fp` library are simple wrappers to API methods that require no storage of information, there are several functions, especially those containing `Get` or `New`, where TestStand is actively allocating new memory to hold the information. In any instance where you are using a function of this type, you must release the allocated memory at the end of the code using calls to `CA_FreeMemory`, `CA_DiscardObjHandle`, or similar functions.

When you are concerned about a function returning a piece of data that must be manually released, refer to the LabWindows/CVI Help or to the NI TestStand Help for that function. Both of these resources explicitly state if the function is allocating memory and often contain additional code fragments explaining how to use the function.

The following are examples of functions that allocate memory:

- `TS_PropertyGetValString()`
- `TS_PropertyGetValIDispatch()`
- `TS_PropertyGetPropertyObject()`
- `TS_NewEngine()`

- `TS_SeqFileNewEditContext()`
- `TS_EngineNewSequence()`

The following example uses one of the previous functions and then releases the memory:

```
char *stringVal = NULL;

TS_PropertyGetValString (propObj, &errorInfo,
"Step.Limits.String", 0, &stringVal);

...

CA_FreeMemory (stringVal);
```

Accessing Built-in Properties in LabWindows/CVI

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

In LabWindows/CVI, you access built-in properties using a property function in the ActiveX driver. The following code obtains the value of the `Sequence.Name` property:

```
int GetSequenceName(CAObjHandle sequence)

{

int error = 0;

ErrMsg errMsg = "";

ERRORINFO errorInfo;

char *sequenceName = NULL;
```

```
tsErrChk(TS_SequenceGetName (sequence, &errorInfo,
&sequenceName));
```

Error:

```
// Free Resources
```

```
if (sequenceName)
```

```
CA_FreeMemory(sequenceName);
```

```
return error;
```

```
}
```

The following function obtains a reference to a step named CVI Pass/Fail Test from a Sequence object:

```
int GetStepInSequence(CAObjHandle sequence)
```

```
{
```

```
int error = 0;
```

```
ErrMsg errMsg = "";
```

```
ERRORINFO errorInfo;
```

```
CAObjHandle step = 0;
```

```
tsErrChk(TS_SequenceGetStepByName (sequence, &errorInfo,
"CVI Pass/Fail Test", StepGroup_Main, &step));
```

Error:

```
// Free Resources
```

```
if (step)
```



```
CA_DiscardObjHandle(step);

return error;

}
```

Accessing Dynamic Properties in LabWindows/CVI

In TestStand, you can define custom step properties, sequence local variables, sequence file global variables, and station global variables. Because the TestStand API is independent of the variables and custom step properties you define, these variables and properties are dynamic with respect to the TestStand API. The TestStand API provides the `PropertyObject` class so you can access dynamic properties and variables from within code modules, where you use lookup strings to identify specific properties by name.

The following example shows how to call a method of the `PropertyObject` class on a handle to a `SequenceContext` object to set a local variable:

```
int SetLocalVariable(CAObjHandle seqContextCVI)

{

    int error = 0;

    ErrMsg errMsg = "";

    ERRORINFO errorInfo;

    VBOOL propertyExists;

    // Set local variable NumericValue to a random number

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

    if (propertyExists)
    tsErrChk(TS_PropertySetValNumber(seqContextCVI, &errorInfo,
```

```
"Locals.NumericValue", 0, rand())));

Error:

return error;

}
```

Programming with TestStand API Constants and Enumerations in LabWindows/CVI

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

The header file for the ActiveX driver defines all constants and enumerations the methods and properties require. The constant and enumeration names start with a prefix, such as `TS_`, followed by the constant or enumeration name.



Note The functions, constants, and enumerations in the `tsapicvi.fp` driver begin with the unique prefix `TS_`. This prefix is not included in the function, constant, and enumeration names in the NI TestStand Help.

For example, the ActiveX driver defines the `RunModes` constant as follows:

```
#define TS_RunMode_Normal "Normal"

#define TS_RunMode_Skip "Skip"

#define TS_RunMode_ForceFail "Fail"

#define TS_RunMode_ForcePass "Pass"
```

The ActiveX driver defines the `StepGroups` enumeration as follows:

```
enum TSEnum_StepGroups
{
    TS_StepGroup_Setup = 0,
    TS_StepGroup_Main = 1,
    TS_StepGroup_Cleanup = 2,
    TS_StepGroupsForceSizeToFourBytes = 0xFFFFFFFF
};
```

For parameters of functions that use an enumeration type, the LabWindows/CVI function panel displays the list of enumerations in a ring control.

For parameters of functions that specify a numeric constant, use the bitwise-OR operator to specify multiple options. For example, the following code sets a local variable only when the variable does not already exist:

```
int options = PropOption_DoNothingIfExists |
PropOption_InsertIfMissing;

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

Programming with the TestStand API in Microsoft Visual Basic .NET

In Microsoft Visual Basic .NET, you can access the TestStand ActiveX API by first adding the `NationalInstruments.TestStand.Interop.API` component as a reference in a project.

To add a reference to the TestStand 4.0 or later API assembly in Visual Studio, select the project in the Solution Explorer. Select **Project»Add Reference** to launch the Reference Manager dialog box. Click **Assemblies»Extensions** and select the TestStand <Version> <APIName> Interop Assembly component from the list. Click **OK** to close the Reference Manager dialog box. Then, right-click the project and select **Properties**, click

the References tab, and select the new reference. In the Properties panel, change the **Embed Interop Types** setting to `False`. If this setting is `True`, using any constants such as `PropertyOptions` in your code will result in compiler errors

If the TestStand Interop assemblies do not appear in the Reference Manager dialog box, exit all running copies of Visual Studio, launch the TestStand Version Selector, select the current version of TestStand, and click the **Make Active** button.

After you add the `NationalInstruments.TestStand.Interop.API` component as a reference in a project, add the following directive at the top of the source file:

```
Imports NationalInstruments.TestStand.Interop.API
```

All TestStand classes, methods, and properties are available in the Microsoft Visual Studio object browser and are accessible from the Visual Basic .NET source code. Help text for classes, methods, and properties appears in Microsoft IntelliSense.

When you create a new project in a supported version of Visual Studio, you must set **Project Properties»Build»Platform Target** to **x86** so the project can access the TestStand API and User Interface (UI) Controls on 64-bit versions of Microsoft Windows.

Related concepts:

- [TestStand API Concepts](#)
- [.NET Support for the TestStand API](#)

Accessing Built-in Properties in Microsoft Visual Basic .NET

In Microsoft Visual Basic .NET, you can access built-in properties by using the native syntax for accessing properties of objects.

The following code obtains the value of the Name property from a Sequence object:

```
Private Sub Example (SequenceObj As Sequence)

Dim NameString As String
```

```
NameString = SequenceObj.Name
```

```
End Sub
```

The following code obtains a reference to a step of a sequence the `Sequence` object references:

```
Private Sub Example (ContextObj As SequenceContext)
```

```
Dim SequenceObj as Sequence
```

```
Dim StepObj as Step
```

```
Set SequenceObj = ContextObj.Sequence()
```

```
Set StepObj = SequenceObj.GetStep(2,  
StepGroups.StepGroup_Main)
```

```
End Sub
```

Accessing Dynamic Properties in Microsoft Visual Basic .NET

In Microsoft Visual Basic .NET, you can access dynamic properties by obtaining a `PropertyObject` reference from the specific object reference using the `AsPropertyObject` method on the object. You can then use the `PropertyObject` interface to access custom properties of the object using a lookup string to specify the specific custom property.

The following code uses the `PropertyObject.GetValString` method to obtain the error message value for the current step:

```
Private sub Example (StepObj as Step)
```

```
Dim PropertyObj as PropertyObject
```

```
Dim ErrorString as String
```

```
Set PropertyObj = StepObj.AsPropertyObject()
```

```
ErrorString =
PropertyObj.GetValString("Result.Error.Msg", 0)

End Sub
```

Acquiring a Derived Class from the PropertyObject Class in Microsoft Visual Basic .NET

In Microsoft Visual Basic .NET, you can acquire a derived class from a dynamic property the PropertyObject class returns.

The following code uses a lookup string to obtain a reference to a Step object from the sequence context:

```
Private Sub Example (ContextObj As SequenceContext)

Dim PropertyObj as PropertyObject

Dim StepObj as Step

Set PropertyObj = ContextObj.AsPropertyObject()

StepObj =
PropertyObj.GetPropertyObject("RunState.Sequence.Main[2]",
0)

End Sub
```

Programming with the TestStand API in C#

In C#, you can access the TestStand ActiveX API by adding the `NationalInstruments.TestStand.Interop.API` component as a reference in a project.

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

Reference Manager dialog box. Then, expand the References node for the project in the Solution Explorer, right-click the new reference and select **Properties** from the context menu. In the Properties panel, change the **Embed Interop Types** setting to `False`. If this setting is `True`, using any constants such as `PropertyOptions` in your code will result in compiler errors

If the TestStand Interop assemblies do not appear in the Reference Manager dialog box, exit all running copies of Visual Studio, launch the TestStand Version Selector, select the current version of TestStand, and click the **Make Active** button.

After you add the `NationalInstruments.TestStand.Interop.API` component as a reference in a project, add the following directive at the top of the source file:

```
using NationalInstruments.TestStand.Interop.API;
```

All TestStand classes, methods, and properties are available in the Microsoft Visual Studio object browser and are accessible from the C# source code. Help text for classes, methods, and properties appears in Microsoft IntelliSense.

To control NI instruments from a .NET assembly, you must also complete the following tasks:

- Install the appropriate NI driver for the instrument you want to control. Ensure you install the correct NI .NET API for the version of the NI driver you installed. For some NI drivers, you must manually select .NET support options in the installer.
- Add a reference to the corresponding NI driver .NET API assembly to the project. The TSM Accelerometer with .NET and the Part Average Testing with .NET examples include Visual Studio projects that reference the TSM Code Module API and some NI .NET APIs and demonstrate how to invoke those APIs. Refer to the ***TestStand Semiconductor Module Manual*** for more information about these examples.

Visit ni.com/info and enter the Info Code `NETAPIdriversupport` for information about the available NI .NET APIs and the versions of the NI drivers each supports.

Upgrading TestStand

When you upgrade to a full, non-service pack version of TestStand and you want to continue to use existing code modules and applications in the later version, you must update the .NET assembly references in Visual Studio projects. The assemblies you build using TestStand APIs are compatible only with the version of TestStand, including service packs, you reference in Visual Studio projects.

Upgrading Instrument Drivers

When you upgrade to a later version of an instrument driver .NET API, you do not need to update the .NET assembly references in Visual Studio projects. NI instrument driver .NET API installers ensure that references to earlier versions of the instrument driver .NET API assembly are automatically redirected to the current version of the assembly if the earlier version assembly does not exist.

Related concepts:

- [TestStand API Concepts](#)
- [.NET Support for the TestStand API](#)

Related information:

- [TestStand Semiconductor Module Manual](#)

Accessing Built-in Properties in C#

In C#, you can access the built-in properties by using the native syntax for accessing properties of objects.

The following code obtains the value of the Name property from a Sequence object:

```
private void Example (Sequence sequenceObj)
{
    String nameString = sequenceObj.Name;
```



```
}
```

The following code obtains a reference to a step of a sequence the `Sequence` object references:

```
private void Example (SequenceContext contextObj)
{
    Sequence sequenceObj;

    Step stepObj;

    sequenceObj = contextObj.Sequence;

    stepObj = sequenceObj.GetStep(2,
    StepGroups.StepGroup_Main);
}
```

Accessing Dynamic Properties in C#

In C#, you can access dynamic properties by obtaining a `PropertyObject` reference from the specific object reference using the `AsPropertyObject` method on the object. You can then use the `PropertyObject` interface to access custom properties of the object using a lookup string to specify the specific custom property.

The following code uses the `PropertyObject.GetValString` method to obtain the error message value for the current step:

```
private void Example (Step stepObj)
{
    PropertyObject propertyObj;

    String errorString;

    propertyObj = stepObj.AsPropertyObject();
```

```

errorString =
propertyObj.GetValString("Result.Error.Msg", 0);

}

```

Acquiring a Derived Class from the PropertyObject Class in C#

In C#, you can acquire a derived class from a dynamic property the `PropertyObject` class returns by using the `as` keyword or by casting.

The following code uses a lookup string to obtain a reference to a `Step` object from the sequence context:

```

private void Example (SequenceContext contextObj)

{

PropertyObject propertyObj;

Step stepObj;

propertyObj = contextObj.AsPropertyObject();

// stepObj would be null if GetPropertyObject did not
return a step

stepObj =
propertyObj.GetPropertyObject("RunState.Sequence.Main[2]", 0)
as Step;

// C# would raise an exception if GetPropertyObject did not
return a step

stepObj =
(Step)propertyObj.GetPropertyObject("RunState.Sequence.Main[2]", 0);

}

```

Programming with the TestStand API in Microsoft Visual C++

None

Using `#import`

You can use the `#import` compiler directive to automatically generate C++ classes for Microsoft Visual Studio to access the objects defined in TestStand type libraries.

To generate C++ classes for TestStand, you must first add the `<TestStand>\Bin` directory to the list of additional include directories in the Visual C++ project settings.

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

In Visual Studio 2010 or later, enable the **Enable TestStand Include Directories** option on the NI TestStand property page of the Project Properties dialog box for a C++ project to specify whether to use TestStand include directories.

After you update the include directories, add the following directive at the top of the source file:

```
#import "teapi.dll"
```

When compiling this directive, Visual C++ generates `teapi.tlh` and `teapi.tli` in a `Debug` or `Release` directory that contains the intermediate files for the project. The generated files contain definitions of C++ classes that correspond to the TestStand API classes. Each C++ class contains methods for each of the corresponding TestStand API methods and properties.

By default, Visual C++ encloses the TestStand-generated classes within a `TS` namespace to avoid possible conflicts with other C++ identifiers that have the same names. As a result, when using the generated classes, you must prefix each class name with `TS::`, as in `TS::Sequence`. To avoid the need to prefix the class names with the namespace, you can add the `no_namespace` attribute to the `import` directive. You can also add a `using namespace TS` statement after the `import` directive, as shown in the following examples:

```
#import "teapi.dll" no_namespace
```

or

```
#import "teapi.dll"
```

```
using namespace TS;
```

In addition to `teapi.dll`, you can import the following files:

- `TsAdpApi.tlb`—Import this type library file to access the TestStand Adapters API. The default namespace for the TestStand Adapters API is `TSAdp`.
- `tsui.dll` and `tsuisupp.dll`—Import these DLLs to access the TestStand User Interface (UI) Controls API. The default namespace for `tsui.dll` is `TSUI` and the default namespace for `tsuisupp.dll` is `TSUISUPP`.

The `tsapivc.h` and `tsapivc.cpp` files import the type libraries for the TestStand API, TestStand Adapters API, and TestStand UI Controls API.

If you plan to use the TestStand API in many different source files, you might want to add the `#import` directive to the precompiled header file. Use `tsapivc.h` and `tsapivc.cpp` to more easily use the TestStand API in many different files. These files are located in the `<TestStand>\API\VC` directory. To use these files, include `tsapivc.h` in the precompiled header and include `tsapivc.cpp` in a single source file in the project so you can access the TestStand API C++ classes in all of the source files that use the precompiled header.

Using Microsoft Foundation Classes (MFC) Wrappers

TestStand does not install MFC wrappers for accessing the TestStand API. If you have source code that uses the `tsapimfc.cpp` and `tsapimfc.h` files from a previous version of TestStand, complete one of the following actions when you upgrade TestStand:

- Convert the code that uses MFC wrappers to use the Microsoft Visual Studio `#import` wrappers instead. Use the `tsapivc.cpp` and `tsapivc.h` files to access the TestStand API using these wrappers.
- Use Visual Studio to generate MFC wrappers for TestStand.
- Continue to use the wrappers provided by a previous version of TestStand.

Related concepts:

- [TestStand Directory Structure](#)

Object References in Microsoft Visual C++/#import

The `#import` directive generates a set of smart pointer C++ classes that simplify the task of maintaining references to objects. The constructors, destructors, and assignment operators in these classes automatically add references to the object and release the object as needed. Each TestStand API class includes a pointer class you can use to maintain a reference to a TestStand object. These pointer classes are named by appending the `Ptr` suffix to the name of the API class to which the pointer refers. For example, the following code declares a pointer to a TestStand Step object:

```
TS::StepPtr step;
```

All of the generated C++ classes use these pointer classes as method parameters and return values.

If a test function or method in a class receives a dispatch pointer as a parameter, declare a local variable of the correct pointer class and assign to it the dispatch pointer. The construction of the pointer adds a reference to the dispatch pointer and the destructor releases it, as shown in the following example:

```
void Example (LPDISPATCH sequenceContextDisp)
{
    TS::SequenceContextPtr context = sequenceContextDisp;

    TS::PropertyObjectPtr contextAsPropObj = context->AsPropertyObject();
}
```

Strings in Microsoft Visual C++/#import

Typically, the classes the `#import` directive generates use the `_bstr_t` class for passing and returning string values. The exception is string output parameters, which

are returned as type `BSTR`. TestStand provides a class called `param_bstr_t` that includes most of the same methods as the `_bstr_t` class and simplifies the task of receiving string output parameters.

Related concepts:

- [Utilities for Microsoft Visual C++/#import](#)

Utilities for Microsoft Visual C++/#import

TestStand provides a set of declarations in the `tsvcutil.h` file, located in the `<TestStand Public>\API\VC` directory, to simplify the task of programming when you use the `#import` directive. Because all identifiers in this header file are enclosed within the namespace `TSUTIL`, you must prefix all uses of the templates and classes with `TSUTIL::`. This file contains the following templates and classes:

- **SafeArray template**—Simplifies the task of programming with SafeArrays and Standard Template Library containers such as vectors. The following example demonstrates how to convert the SafeArray of `PropertyObjectFile` objects the `Engine.GetTypeUsageLocations` method returns into a vector of `PropertyObjectFilePtr` pointers:

```
#include "tsapivc.h"

#include "tsvcutil.h"

#include <vector>

void GetFilesUsingTypeName(TS::IEngine *engine, const
char *typeName, std::vector<TS::PropertyObjectFilePtr>
&filesVector)

{

SAFEARRAY *fileSAFEARRAY;

// Get the SAFEARRAY of PropertyObjectFiles
```

```

fileSAFEARRAY = engine->GetTypeUsageLocations(typeName);

// Create a SafeArray object to take ownership of the
array returned above.

TSUTIL::SafeArray<LPDISPATCH,VT_DISPATCH> fileArray;

fileArray.Set(fileSAFEARRAY, true);

// Put the SafeArray into a vector

fileArray.GetVector(filesVector);

}

```

The following example shows how to convert the SafeArray of strings the Engine.DisplayFileDialog method returns into a vector of `_bstr_t` strings:

```

#include "tsapivc.h"

#include "tsvcutil.h"

#include <vector>

void DisplayFileDialog(TS::IEngine *engine,
std::vector<_bstr_t> &selectedPathsVector)

{

TSUTIL::SafeArray<BSTR, VT_BSTR> selectedPathsSafeArray;

TSUTIL::SafeArray<BSTR, VT_BSTR> absolutePathsSafeArray;

VARIANT_BOOL gotPath;

gotPath = engine->DisplayFileDialog("" /* dialog title
*/, "" /* ok button */, "" /* initial path */,

&selectedPathsSafeArray, &absolutePathsSafeArray,

```

```

TS::OpenFile_UseAbsolutePath |
TS::OpenFile_HideMultiSelectListCtrl, "seq",

TS::WinFileDlg_HIDEREADONLY |
TS::WinFileDlg_FILEMUSTEXIST |
TS::WinFileDlg_ALLOWMULTISELECT,

"SequenceFiles|*.seq||", vtMissing /* current file */,
&vtMissing, &vtMissing);

if (gotPath)

absolutePathsSafeArray.GetVector(selectedPathsVector);

}

```

- **param_bstr_t class**—Receives strings from output parameters, as shown in the following example:

```

TS::PropertyValueTypes elemType;

long numElements;

TSUTIL::param_bstr_t lowerBounds;

TSUTIL::param_bstr_t upperBounds;

array->GetDimensions("", 0, &lowerBounds, &upperBounds,
&numElements, &elemType);

```

- **IsOfType template**—Determines whether a TestStand object implements a specific API class. For example, the following code determines whether the object the PropertyObject pointer objectPtr references is actually a Step object:

```

TS::PropertyObjectPtr objectPtr;

bool objectIsStep =
(IsOfTypeFunctor<TS::Step>::ObjectIsOfType(*objectPtr));

```


Related concepts:

- [TestStand Directory Structure](#)

Accessing Built-in Properties in Microsoft Visual C++/#import

In Microsoft Visual C++, the `#import` directive generates methods for accessing the values of the built-in properties for each TestStand class. The following C++ code obtains the value of the Name property from a Sequence object:

```
void Example (LPDISPATCH sequenceDisp){

TS::SequencePtr sequence = sequenceDisp;

_bstr_t nameString = sequence->GetName();

}
```

The following code obtains a reference to a step in a sequence the `Sequence` object references:

```
void Example (LPDISPATCH sequenceContextDisp){

TS::SequenceContextPtr sequenceContext =
sequenceContextDisp;

TS::SequencePtr sequence;

TS::StepPtr step;

sequence = sequenceContext->GetSequence();

step = sequence->GetStep(2, TS::StepGroup_Main);

}
```

Accessing Dynamic Properties in Microsoft Visual C++/#import

In Microsoft Visual C++, you can use the classes the `#import` directive generates to

access dynamic properties by obtaining a `PropertyObject` reference from the specific object reference using the `AsPropertyObject` method on the object. You can use the `PropertyObject` interface to access custom properties of the object using a lookup string to specify the custom property.

The following code uses the `PropertyObject.GetValString` method to obtain the error message value for the current step:

```
void Example (LPDISPATCH seqContextDisp){

    TS::SequenceContextPtr sequenceContext = seqContextDisp;

    TS::PropertyObjectPtr property;

    _bstr_t errorString;

    property = sequenceContext->GetStep()->AsPropertyObject();
    errorString = property->GetValString("Result.Error.Msg", 0);

}
```

Acquiring a Derived Class from the PropertyObject Class in Microsoft Visual C++/#import

In Microsoft Visual C++, you can use a class the `#import` directive generates to acquire a derived class from a dynamic property the `PropertyObject` class returns.

The following code uses a lookup string to obtain a reference to a `Step` object from the sequence context:

```
void Example (LPDISPATCH seqContextDisp){

    TS::SequenceContextPtr sequenceContext = seqContextDisp;

    TS::PropertyObjectPtr property;

    TS::StepPtr step;
```

```
property = sequenceContext->AsPropertyObject();

step = property-
>GetPropertyObject("RunState.Sequence.Main[2]", 0);

}
```

Programming with the TestStand API in HTBasic

HTBasic uses a CSUB library TestStand provides to access the TestStand ActiveX API. The CSUB library contains HTBasic subroutines for the `GetVal` and `SetVal` methods in the TestStand API.

Accessing Property Values in HTBasic

You can access the values of built-in properties using `Get` and `Set` subroutines in the TestStand API CSUB library. The name of each `Get` or `Set` subroutine starts with `Getval` or `Setval`, followed by the data type name.



Note Some type names are abbreviated because of HTBasic naming limitations.

Function Name	TestStand Type	HTBasic type
Getvalnumber	Number	REAL
Setvalnumber	Number	REAL
Getvalstring	String	STRING
Setvalstring	String	STRING
Getvalboolean	Boolean	INTEGER
Setvalboolean	Boolean	INTEGER
Getvalnumarray	Array of Number	Array of REAL
Setvalnumarray	Array of Number	Array of REAL
Getvalstrarray	Array of String	Array of STRING
Setvalstrarray	Array of String	Array of STRING

The following example subroutine demonstrates how to access step properties. The subroutine obtains the value of the upper limit in the current step and sets the measurement value to the upper limit.

```
SUB Test

REAL Realnum
CALL Getvalnumber("Step.Limits.High",0,Realnum)
CALL Setvalnumber("Step.Result.Numeric",0,Realnum)

SUBEND
```

Using Strings in HTBasic

You must choose HTBasic string variables that are large enough to contain the string property values from TestStand. If the HTBasic variable is not large enough, TestStand truncates the value of the string you obtain.



Note Strings you pass to TestStand have no size restrictions.

Using Arrays in HTBasic

You can use the CSUB library subroutines to exchange one- or two-dimensional arrays between TestStand and HTBasic. When retrieving an array from TestStand, you must choose HTBasic array variables as large as the TestStand array. If the HTBasic array is smaller, TestStand returns the ERROR 16 Improper or Inconsistent Dimensions error.

When retrieving a string array, you must ensure that the HTBasic strings are large enough to contain the TestStand strings. If the HTBasic strings are too short, TestStand truncates the string values. When you pass an array to TestStand, TestStand automatically adjusts its array to match the dimensions of the HTBasic array, as shown in the following example:

```
SUB Test2dstr

DIM Strings$ (2, 5) [30]
CALL Getvalstrarray ("Local.StringArray2",1,Strings$ (*) )

SUBEND
```

The following example shows the equivalent TestStand declaration for the numeric array, assuming that the HTBasic OPTION BASE is zero: `StringArray2 Array of Strings [0 .. 2] [0 . . 5]`

Related concepts:

- [Accessing Arrays Using API - HTBasic](#)
- [Accessing Properties Using API](#)

.NET Support for the TestStand API

TestStand installs .NET interop assemblies for the TestStand API in the `<TestStand>\API\DotNet\Assemblies` directory and in the Global Assembly Cache (GAC). The interop assemblies support the current and earlier versions of the TestStand API, as shown in the following table.

TestStand API Version		.NET Version
TestStand 2025 Q2 and later		.NET 8 (CLR 5.0 or later)
TestStand 2024 Q4	For .NET Adapter	.NET 8 (CLR 5.0 or later)
	For user interfaces	.NET Framework CLR 4.0
TestStand 2012 — TestStand 2023 Q4		.NET Framework CLR 4.0
TestStand 4.0 — TestStand 2010 SP1		.NET Framework CLR 2.0
TestStand 3.5 or earlier		.NET Framework CLR 1.1

Writing an Application with the TestStand User Interface Controls

You can create or customize a user interface application using the TestStand User Interface (UI) Controls based on the various example user interface applications TestStand provides.

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

Related concepts:

- [Creating Custom User Interfaces](#)

Writing an Application with the TestStand Engine API

You can use the TestStand Engine API to write a complete application.



Note NI strongly recommends using the TestStand User Interface (UI) Controls to write a TestStand application.

Related concepts:

- [Writing an Application with the TestStand User Interface Controls](#)

Creating the TestStand Engine

When you write an application using the TestStand API, you must create a TestStand Engine object. Because you can create the TestStand Engine as an ActiveX control or as an ActiveX Automation server, you can access the engine from any application development environment that supports ActiveX. If you use a development environment that supports ActiveX controls with events, such as Microsoft Visual Basic or Microsoft Visual C/C++, create the `Engine` object by adding the TestStand Engine control to the main form of the application. If the development environment does not support ActiveX controls, you can create the `Engine` object as an in-process automation server.



Note NI recommends that you call the `Engine.LoadTypePaletteFilesEx` method when you create the TestStand Engine to ensure that the application loads the correct versions of step types and data types. If you do not load the type palettes and load a sequence file with older versions of NI types, the sequence file might not execute correctly.

User Interface Messages (UIMessages)

When you write an application that runs TestStand sequence files, you do not handle

the details of executing each step. Instead, you call the TestStand Engine to begin executing the sequence file in a new thread. The engine notifies you asynchronously of the state of the execution by sending you messages such as Start, End, Break, and Trace. You can update the application user interface in response to these messages.

You can create an execution directly or indirectly. For example, you create an execution indirectly when you load a sequence file that has a load callback. Before you create an execution directly or indirectly, you must create a user interface message handler. The TestStand Engine uses user interface messages to communicate the execution states and other asynchronous information to the application.

Creating a User Interface Message Handler

If you create the TestStand Engine as an ActiveX control, the easiest way to handle user interface messages is to implement the `UIMessageEvent` callback sequence for the ActiveX control. Whenever the engine sends a user interface message for an application, the engine calls the callback with a `UIMessage` object as a parameter.



Note When you use a programming language such as C# or Microsoft Visual Basic .NET that uses garbage collection to dispose of objects, you must explicitly acknowledge messages when you handle user interface messages by calling the `UIMessage.Acknowledge` method. If you do not call the `UIMessage.Acknowledge` method, the TestStand Engine cannot send new messages until the garbage collector disposes of the `UIMessage` object for the previous message, and the application hangs.

If you do not create the engine as an ActiveX control and are not using the TestStand User Interface (UI) Controls, you can handle user interface messages in the following two ways:

- When you can write a C-callable function in the application development environment (ADE), you can instruct the engine to call such a function by passing its address to the `Engine.RegisterUIMessageCallbackEx` method. The application thread that calls this method must regularly process Microsoft Windows messages to use this option.
- When the ADE cannot create C-callable functions, you must complete the following steps to create a polling loop for user interface messages.

1. Inform the engine that you are polling by setting the `Engine.UIMessagePollingEnabled` property to `True`.
2. Implement a polling loop by periodically checking the `Engine.IsUIMessageQueueEmpty` property.
3. If the queue is not empty, call the `Engine.GetUIMessage` method to obtain the next user interface message from the engine. Do not check the queue again until you complete whatever action the user interface message requires and release all references to the `UIMessage` object.



Tip One way to implement a polling loop in an application is to create a Timer callback that polls each time it executes.

Handling Specific User Interface Messages

The first thing to do when you receive a user interface message is to check its `UIMessage.Event` property, the value of which is a `UIMessageCode`. Use the `UIMessageCode` to determine which items, if any, to update on the user interface.

Execution Events

Most user interface messages notify you to update an Execution window. When you receive one of these user interface messages, use the `UIMessage.Execution` property to obtain a reference to the Execution object to which the message applies. Use the `Execution.Id` property to determine which user interface display to update.

The following user interface messages apply to executions:

- **Start/End Events**—If you receive a `UIMsg_StartExecution` event, create a new window for the execution or reinitialize an existing window for the execution. The Execution window already exists when the user restarts the execution. If you receive a `UIMsg_EndExecution` event, modify the display for the execution to show that the execution has ended. Then, release any handles to `Thread`, `Step`, or `SequenceContext` objects you have for the execution, and display the execution report, if one exists.

TestStand posts the `UIMsg_StartInteractiveExecution` and `UIMsg_EndInteractiveExecution` events to notify you when the user runs steps

interactively in an execution. The user can run steps interactively in a new execution or from an existing execution while at a breakpoint.

TestStand posts the `UIMsg_StartFileExecution` and `UIMsg_EndFileExecution` events to notify the user interface of the period during which an execution uses a particular sequence file. You can use this information to disable the editing of the sequence file or to indicate that it is in use.

- **Break Events**—TestStand posts the `UIMsg_BreakOnBreakpoint`, `UIMsg_BreakOnUserRequest`, or `UIMsg_BreakOnRunTimeError` user interface message when an execution enters a suspended state. Each of these messages indicates a different reason for the change in state.

For each of these events, you can choose to display the following information:

- Information about the steps in the currently executing step group
- A call stack list that identifies the locations of the active sequence calls in the current thread
- A list of the threads the execution contains

To display information about the steps in the currently executing step group, you must first obtain the foreground Thread object by obtaining the value of the `Execution.ForegroundThreadIndex` property and then calling the `Execution.GetThread` method with the index. From the `Thread` object, you can then obtain the `SequenceContext` object that represents the top of the call stack by calling the `Thread.GetSequenceContext` method. Pass 0 for the call stack index and then pass a variable in which to store the frame ID, which is a unique identifier for the call stack entry. The frame ID is not required to handle this event, but is useful when handling .

Once you have the `SequenceContext` object, you can access all the information about the currently executing sequence. The `SequenceContext.Sequence` property provides the run-time copy of the sequence. Access this run-time copy to obtain all the information about the steps you require. You can also use the other `SequenceContext` properties to obtain information such as the currently executing step group, the sequence file that contains the executing sequence, and other information.

To display a call stack list, first obtain the `Thread.CallStackSize` property of the

current thread to determine the number of call stack entries. Then obtain the `SequenceContext` object for each entry by passing its zero-based index to the `Thread.GetSequenceContext` method. Use the `SequenceContext.CallStackName` property to obtain the display name for an entry.

To display a thread list, first obtain the number of threads from the `Execution.NumThreads` property. Then, obtain each thread by passing its zero-based index to the `Execution.GetThread` method. Obtain the display name of the thread from the `Thread.DisplayName` property.

When you receive the `UIMsg_BreakOnRunTimeError` event, display the run-time error to the user after updating the user interface. Obtain the run-time error message from the `SequenceContext.GetRunTimeErrorMessageEx` method for the top-level `SequenceContext`. Resuming execution from a `UIMsg_BreakOnRunTimeError` event causes the execution to go to the Cleanup step group. To ignore the run-time error, you must first call the `Thread.ClearCurrentRTE` method of the foreground thread. However, for trace events, call the `Engine.DisplayRunTimeErrorDialogEx` method to launch the Run-time Error dialog box. You can also create a custom dialog box to launch instead.

- **Trace Events**—The update to the user interface a `UIMsg_Trace` event requires is very similar to that of a break event, except that you use only the name of the foreground thread and the top entry in the call stack list instead of the names of all the threads and call stack entries because the user interface disallows switching threads or browsing the call stack while tracing. In addition, because the trace event does not indicate a suspended state, you do not need to call the `Execution.Resume` method to continue. Instead, execution continues automatically when you finish handling the user interface message for the trace event by releasing the user interface message or returning from the event handling callback sequence. In addition, you can use the **frameId** output parameter of the `Thread.GetSequenceContext` method to update the display.

As an optimization, the user interface holds the step group and **frameId** value from the last trace event. If the next event is a trace event and its sequence context specifies the same **frameId** value and step group, you need only to update the display of the previously executed step and the step to execute next. Because this is typically the case when users trace through a long list of steps in the same sequence and step group, this optimization can significantly increase the

maximum tracing speed of the application.



Note If the application polls for user interface messages, the rate at which you poll limits the maximum tracing rate.

- **Resume Events**—TestStand posts the `UIMsg_ResumeFromBreak` event to the user interface whenever an execution is about to resume from a suspended state. This event allows the user interface to update the enabled state of controls on its execution display and to show that the execution is running.
- **Execution State Change Events**—The purpose of the execution state change events is to notify the user interface that the state of an execution is about to change. The action these events specify does not occur until you finish handling the user interface message by releasing the user interface message or returning from the event handling callback. also signify an execution state change.

The name of the following execution state change events indicate the meaning for each event:

- `UIMsg_TerminatingExecution`
- `UIMsg_TerminationCancelled`
- `UIMsg_TerminatingInteractiveExecution`
- `UIMsg_AbortingExecution`
- `UIMsg_KillingExecutionThreads`
- **Progress Events**—A step posts progress events to notify the user interface to update its progress indicator or text message, if one exists. If you receive a `UIMsg_ProgressPercent` event, check the `UIMessage.NumericData` property for the percentage of the progress indicator to shade. If you receive a `UIMsg_ProgressText` event, you can find the text to display in the `UIMessage.StringData` property.

Related concepts:

- [Posting User Interface Messages](#)

Shutdown Events

TestStand generates the `UIMsg_ShutDownComplete` or the `UIMsg_ShutDownCancelled` event after you call the `Engine.ShutDown` method.

This event informs you that the shutdown request has ended.

Related concepts:

- [Shutting Down the Engine](#)

User-Defined Events

You can post a custom message in a user-defined event by using an event number greater than or equal to the value of the `UIMsg_UserMessageBase` constant.

Related concepts:

- [Posting User Interface Messages](#)

Handling Unknown Events

If you receive a user interface message with an event ID the application does not expect, complete one of the following steps:

- If you are polling, release the user interface message and continue polling.
- If you receive the user interface message in a callback, return from the callback without responding to the message.



Note Do not treat user interface messages with unexpected event IDs as errors. Ignoring unknown events ensures that the application is compatible with future versions of TestStand and that it behaves consistently with unknown user-defined events.

Posting User Interface Messages

TestStand currently supports posting user interface messages by calling the `Thread.PostUIMessageEx` method, which allows the steps of an execution to post progress events and user-defined events. Because the `PostUIMessageEx` method is a method of the `Thread` class, the thread and its associated execution are automatically properties of the user interface message. You can send additional data using the **numericDataParam** and **stringDataParam** parameters of the method.

Related concepts:

- [Execution Events](#)
- [User-Defined Events](#)

Managing Sequence Files in an Application

None

Loading and Caching

To obtain a reference to a sequence file on disk, call the `Engine.GetSequenceFileEx` method. If the sequence file is already in memory, the method returns a reference to the file in the cache. If the sequence file is not already in memory, this method loads the sequence file from disk and places it in the internal sequence file cache of the TestStand Engine. Every sequence file reference you obtain from the `Engine.GetSequenceFileEx` method causes the cache to increment an internal usage count for the sequence file. When you are finished with the sequence file, you must call the `Engine.ReleaseSequenceFileEx` method to decrement the usage count. If your development environment requires you to explicitly release COM references, you must call the `Engine.ReleaseSequenceFileEx` method before releasing the COM reference to the sequence file.

To create a new sequence file, call the `Engine.NewSequenceFile` method, which adds the new sequence file to the internal sequence file cache of the engine. Therefore, before you release a sequence file you obtain from the `Engine.NewSequenceFile` method, you must call the `Engine.ReleaseSequenceFileEx` method on the sequence file. The new sequence file contains one sequence named `MainSequence`. You can .

You can also use the `SequenceFile.AddLoadReference` method to add an additional load reference to a sequence file. You must call the `Engine.ReleaseSequenceFileEx` method to decrement the load reference.

Structure of a Sequence File

A sequence file contains an array of sequences. Each sequence contains Setup, Main, and Cleanup step groups. To access the sequences in a sequence file, use the following

methods and properties of the SequenceFile object:

- NumSequences property
- GetSequence method
- GetSequenceByName method
- GetSequenceIndex method

To access the steps in a sequence, use the following methods of the Sequence object:

- GetNumSteps method
- GetStep method
- GetStepByName method
- GetStepIndex method

Notice that all of the functions for accessing steps use a StepGroups constant to specify the step group that contains the step.

Adding New Steps and Sequences

Every step you insert into a sequence must be a unique instance. If you want to use the same step in two sequences, you must insert separate copies of the step. If you want to repeat a step in a sequence, you must insert a separate copy of the step. You can make a copy of a step by calling the PropertyObject.Clone method. Thus, you can insert a step after you create it, clone it, or remove it from a sequence. This functionality also applies to inserting sequences into sequence files.

To create a new step, call the Engine.NewStep method and then pass an adapter key name and a step type name using the AdapterKeyNames and StepTypes constants. You can then insert the new step into a sequence using the Sequence.InsertStep method. Name the new step and specify its adapter and step-type-specific properties as necessary.

To create a new sequence, call the Engine.NewSequence method and insert the new sequence into a sequence file using the SequenceFile.InsertSequenceEx method. Name the new sequence with a name that is unique among the sequences in the sequence file.

Other methods you can use to implement sequence file editing are

SequenceFile.RemoveSequence, SequenceFile.DeleteSequence, SequenceFile.SequenceNameExists, Sequence.RemoveStep, Sequence.DeleteStep, and Sequence.StepNameExists.

Managing Process Model Entry Points

This topic requires a thorough understanding of process models and how you use them.

Applications typically display the entry points the process model defines as menu items or buttons the user can select.

Obtaining the Process Model Sequence File

Complete the following steps to first obtain the process model sequence file to determine which process model entry points apply to a given situation.

1. If you are displaying an execution, call the `Execution.GetSequenceFile` method of the execution and then call the `SequenceFile.GetModelSequenceFile` method on the sequence file the `Execution.GetSequenceFile` method returns.

If the `SequenceFile.GetModelSequenceFile` method returns a `NULL` reference, continue to step 3. The sequence file the `Execution.GetSequenceFile` method returns is the client sequence file.

2. If you are displaying a sequence file, call the `SequenceFile.GetModelSequenceFile` method on the sequence file.

If the `SequenceFile.GetModelSequenceFile` method returns a `NULL` reference, continue to step 3. The sequence file you display is the client sequence file.

3. If you are not displaying a sequence file or an execution, or if the execution or sequence file does not specify a model sequence file, call the `Engine.GetStationModelSequenceFile` method.

If the `Engine.GetStationModelSequenceFile` method returns a `NULL` reference, no model sequence file currently applies. Otherwise, the

`Engine.GetStationModelSequenceFile` method returns the applicable model sequence file. In this case, no client sequence file exists.

Obtaining the Applicable Entry Point Sequences

If no model sequence file applies to the current context in the user interface, no entry points currently apply. If a model sequence file does apply, you can determine which, if any, sequences in the file are entry points and whether they apply by enumerating all the sequences in the model sequence file. First, obtain the number of sequences from the `SequenceFile.NumSequences` property of the model sequence file. Then, obtain each individual sequence using the `SequenceFile.GetSequence` method. To obtain each sequence, pass indices 0 through (`NumSequences - 1`). Complete the following steps to determine if each sequence is an applicable entry point.

1. Is the `Type` property of the sequence equal to the value of the `SeqType_CfgEntryPoint` enumeration? If so, continue to step 5. If not, continue to step 2.
2. Is the `Type` property of the sequence equal to the value of the `SeqType_ExeEntryPoint` enumeration? If so, continue to step 3. If not, this sequence is not an applicable entry point.
3. Did you obtain the model sequence file from a client sequence file? If so, continue to step 4. If not, this sequence is not an applicable entry point. Notice that the latter case applies when you obtain the model sequence file by calling the `Engine.GetStationModelSequenceFile` method.
4. Does the client sequence file from which you obtained the model sequence file contain a sequence called `MainSequence`? If so, continue to step 5. If not, this sequence is not an applicable entry point.
5. If the `Sequence.ShowEntryPointForEditorOnly` property is `False` or the application is a sequence editor, continue to step 6. If the `Sequence.ShowEntryPointForEditorOnly` property of the sequence is `True` and the application is not a sequence editor, this sequence is not an applicable entry point.
6. If the `Sequence.ShowEntryPointForExeWindow` property of the sequence is `True` and you are displaying an Execution window, this sequence is an applicable entry point. Add it to the list of entry point sequences. If not, continue to step 7.
7. If the `Sequence.ShowEntryPointForFileWindow` property of the sequence is `True` and you are displaying a sequence file, this sequence is an applicable entry point.

Add it to the list of entry point sequences. If not, continue to step 8.

8. If the `Sequence.ShowEntryPointForAllWindows` property of the sequence is `True`, this sequence is an applicable entry point. Add it to the list of entry point sequences. If not, this sequence is not an applicable entry point.

You now have a list of applicable entry point sequences. The next step is to create the menu items and/or command buttons that allow the user to create executions from the entry points.

Creating the Menu Items

When creating an entry point, you must determine which menu or submenu in which to place it. You can place entry points by type, such as putting entry points of the `SeqType_CfgEntryPoint` type in the Configure menu and entry points of the `SeqType_ExeEntryPoint` type in the Execute menu, or you can use the `Sequence.GetEntryPointMenuFromHint` method of the entry point sequence. If the `Sequence.GetEntryPointMenuFromHint` method cannot find a menu that matches any of the menu hints for the entry point, place the entry point menu item by type.

After you determine the location of the menu item, obtain the string to display for the menu item by calling the `Sequence.EvalEntryPointNameExpressionEx` method of the entry point sequence. Pass a reference to the client sequence file as the parameter to the method, or pass a `NULL` reference if no client sequence file exists. Similarly, call the `Sequence.EvalEntryPointEnabledExpressionEx` method to determine whether to enable the menu item.



Note In addition to checking the value the `Sequence.EvalEntryPointEnabledExpression` method returns, you might also want to check certain user privileges, such as `Priv_Execute` or `Priv_RunAnySequence`.

Executing an Entry Point

Consider the following situations for executing an entry point. If a client sequence file exists, you must first create an `EditArgs` object by calling the `Engine.NewEditArgs` method. Use the `EditArgs` object to specify which TestStand objects the user selects

in the application. If the client sequence file is from an execution, call the `EditArgs.SetSelectedExecution` method of the newly created `EditArgs` object and pass a reference to the execution. If the client sequence file is from a Sequence File window, call the `EditArgs.SetSelectedSequenceFile` method and then call the `EditArgs.AddSelectedSequence` method and the `EditArgs.AddSelectedStep` method to add any currently selected steps and sequences.

Pass the following parameters as arguments to the `Engine.NewExecution` method:

Parameter Name	What to Pass
<code>sequenceFileParam</code>	A reference to the client sequence file, or a <code>NULL</code> reference if no client sequence file exists. Depending on the application development environment you use, pass <code>0</code> or the <code>Nothing</code> keyword to indicate a <code>NULL</code> reference.
<code>sequenceNameParam</code>	The value of the <code>Sequence.Name</code> property of the entry point sequence.
<code>processModelParam</code>	A reference to the model sequence file.
<code>breakAtFirstStep</code>	<code>False</code> .
<code>executionTypeMaskParam</code>	If the value of the <code>Sequence.EntryPointInitiallyHidden</code> property of the entry point sequence is <code>True</code> , pass the value of the <code>ExecTypeMask_InitiallyHidden</code> constant. Otherwise, pass the <code>ExecTypeMask_Normal</code> constant.
<code>sequenceArgsParam[Optional]</code>	Leave this parameter unspecified.
<code>editArgsParam[Optional]</code>	Specify this parameter only if a client sequence file exists. Pass a reference to the <code>EditArgs</code> object you create according to the instructions earlier in this section.
<code>InteractiveArgsParam[Optional]</code>	Specify this parameter only if you want to create an interactive execution.

Related concepts:

- [Process Model Architecture](#)
- [Enforcing User Privileges](#)
- [Starting an Interactive Execution in an Application](#)

Implementing a Tools Menu

To implement a Tools menu, you must first call the `Engine.GetRunTimeToolMenuItems` method, which returns a collection of `RunTimeMenuItem` objects. Each object in the collection represent menu items or submenus to display. The `Engine.GetRunTimeToolMenuItems` method takes an `EditArgs` object as a parameter. Pass an `EditArgs` object to indicate which execution, sequences, and steps are selected when the user requests to display the Tools menu.

For each menu item, the `RunTimeMenuItem.Text` property specifies the menu item text and the `RunTimeMenuItem.ItemEnabled` property specifies whether the menu item is enabled. If the `RunTimeMenuItem.SubMenuItems` property is not `NULL`, you must display the menu item as a submenu. The `RunTimeMenuItem.SubMenuItems` property returns a collection of its submenu items.

Use `RunTimeMenuItem.InvokeItem` to execute a menu item, which launches an executable or runs a sequence in a new execution. Pass the same `EditArgs` object as the parameter you used when calling the `Engine.GetRunTimeToolMenuItems` method.

In addition to displaying the Tools menu items in an application, you can also add a Customize menu item to allow the user to configure the Tools menu. To do so, add a menu item to the Tools menu that calls the `Engine.DisplayToolMenuDialog` method. To customize the Tools menu programmatically, operate on the collection the `Engine.GetEditTimeToolMenuItems` method returns.

Executing a Sequence in an Application

The most direct way to create an execution is to execute a specific sequence in a specific sequence file. Although this method does not use a process model, it provides additional control for debugging. To execute a particular sequence, call the `Engine.NewExecution` method and pass the arguments listed in the following table.

Parameter Name	What to Pass
<code>sequenceFileParam</code>	A reference to the sequence file that contains the sequence to execute.

Parameter Name	What to Pass
sequenceNameParam	A string that contains the name of the sequence to execute.
processModelParam	A <code>NULL</code> reference. Depending on the application development environment you use, pass <code>0</code> or the <code>Nothing</code> keyword to indicate a <code>NULL</code> reference.
breakAtFirstStep	<code>False</code> .
executionTypeMaskParam	The value of the <code>ExecTypeMask_Normal</code> constant.
sequenceArgsParam[Optional]	Leave this parameter unspecified.
editArgsParam[Optional]	Leave this parameter unspecified.
InteractiveArgsParam[Optional]	Leave this parameter unspecified.

Starting an Interactive Execution in an Application

You can start an interactive execution from an execution suspended at a breakpoint using the `Thread.DoInteractiveExecution` method or as a new execution using the `Engine.NewExecution` method.

Creating the InteractiveArgs Object

When you start an interactive execution, create an `InteractiveArgs` object and then pass it to the `Thread.DoInteractiveExecution` method or the `Engine.NewExecution` method. Use the `InteractiveArgs` object to specify which steps to run and whether and how the execution loops.

To determine whether and how the user of the application wants the execution to loop, use the dialog box that launches when you call the `Engine.DisplayLoopOnStepsDialog` method or write a custom dialog box to launch instead. Next, call the `Engine.NewInteractiveArgs` method to create the `InteractiveArgs` object. Set the `InteractiveArgs.StepGroup`, `InteractiveArgs.LoopCount`, and `InteractiveArgs.StopExpression` properties of the object as needed. You do not have to set the `InteractiveArgs.LoopCount` or `InteractiveArgs.StopExpression` properties if you do not want the interactive execution to loop. Finally, call the `InteractiveArgs.AddStepIndex` method for each step you want to run, passing the zero-based step index for each step. In an interactive execution, you can only run steps located in the same sequence and step

group. You can then pass the `InteractiveArgs` object to the `Engine.NewExecution` method or the `Thread.DoInteractiveExecution` method.

Starting as a New Execution

Create a new interactive execution in the same manner as you would create a new execution to execute a specific sequence, but instead pass an **InteractiveArgs** parameter. Create an `InteractiveArgs` object and then pass it to the `Engine.NewExecution` method.

Starting From a Suspended Execution

If you are currently displaying an execution that is suspended at a breakpoint, you can start an interactive execution on steps in the current step group and in the current thread of the execution by calling the `Engine.DoInteractiveExecution` method on the current thread with an `InteractiveArgs` object.



Note You cannot start an interactive execution from a breakpoint in another interactive execution.

Related concepts:

- [Executing a Sequence in an Application](#)

Log in/Log out Execution

The log in and log out procedure is implemented as a Front-End callback sequence, in which a sequence you call from the application contains the code that prompts the user for a login and processes the login. Call the `LoginLogout` Front-End callback when the application starts, when you want to give the user the ability to log in or log out, and when you want the user to log out before you shut down the engine. You call the `LoginLogout` callback by using the `Engine.CallFrontEndCallbackEx` method and passing `LoginLogout` for the **sequenceNameParam** parameter.

The `LoginLogout` callback logs out the currently logged-in user. Depending on the value of the arguments in the **argumentList** parameter, the `LoginLogout` callback can

also log in a new user.

ArgumentList Parameter

The **argumentList** parameter to the `Engine.CallFrontEndCallbackEx` method is a property object that contains a set of subproperties. Each subproperty represents an argument. To create the `PropertyObject` object for the **argumentList** parameter, call the `Engine.NewPropertyObject` method as follows:

```
NewPropertyObject (PropValType_Container, False, "", 0)
```

Call the `PropertyObject.SetValBoolean` method on the newly created `PropertyObject` to specify the value of the first argument, `logoutOnly`. You can choose a `True` or a `False` setting, as shown in the following examples:

- `SetValBoolean ("logoutOnly", PropOption_InsertIfMissing, True)`
- `SetValBoolean ("logoutOnly", PropOption_InsertIfMissing, False)`

Finally, call the `PropertyObject.SetValBoolean` method to specify the value of the second argument, `isInitialLogin`. This argument indicates whether this is the initial login at application startup. The callback uses this information to implement the Automatically Login Windows System User feature for Microsoft Windows.

After you call the `Engine.CallFrontEndCallbackEx` method, release the `PropertyObject` you created for the **argumentList** parameter.

Special Handling of the User Interface Message `UIMsg_EndExecution`

To determine when a log out or log in/log out operation has completed, keep track of the `Execution.Id` property of the execution the

`Engine.CallFrontEndCallbackEx` method returns. When you receive the `UIMsg_EndExecution` user interface message, you can use this handle to determine whether the completed execution is the Front-End callback you created. If you know when the LoginLogout callback completes, you can perform the following tasks:

- Update the display to show the name of the currently logged-in user. Obtain the

User object for the current user from the Engine.CurrentUser property and then obtain the name of the user from the User.LoginName property. You can also use the User.FullName property if you prefer.

- Process command-line arguments after a user has logged in.



Tip Do not load a sequence file or run an execution in response to command-line arguments until you give the user a chance to log in.

- Shut down after the last user logs out.
- Update the state of menu items and other controls that depend on user privileges. You must re-evaluate menu items and other controls that relate to process model entry points. The results of the name expressions and enabled expressions for entry points can vary depending on the current user.

Related concepts:

- [Modifying Front-End Callbacks](#)
- [Shutting Down the Engine](#)

Shutting Down the Engine

You shut down the engine just before you exit the application and when you log out the current user. Shutting down the engine is a two-pass process in which you release sequence files and wait for their unload callbacks to complete.



Note

- NI recommends that you call the ApplicationMgr.Shutdown method to shut down the engine for an application you develop with the TestStand User Interface (UI) Controls.
- For a .NET user interface, refer to the Additional Considerations for .NET Application to Properly Shutdown TestStand section of the Starting and Shutting Down TestStand from a .NET User Interface topic for more information about properly starting and shutting down TestStand from .NET applications.

First Pass

Complete the following tasks in the first pass of this process.

- Unload all sequence files and close all completed executions. The application must release all references to sequence files and executions so TestStand can unload all associated sequence files and run any unload callbacks they include.
- Call the `Engine.ShutDown` method and pass `False` for the **final** parameter.

The first pass ends when you receive the `UIMsg_ShutDownComplete` or `UIMsg_ShutDownCancelled` `UIMessage`. If you receive the `UIMsg_ShutDownComplete` event, continue with pass two. If you receive the `UIMsg_ShutDownCancelled` event instead, resume the normal operation of the application. The user can cancel shutdown by clicking **Cancel** in the Waiting for Execution to Complete Timeout dialog box, which TestStand launches for executions that do not end within their allotted time.

Second Pass

Complete the following tasks in the second pass of this process.

- Close all executions TestStand created as a result of the first pass. Release references to all executions and close the windows you used to display them.
- Call the LoginLogout Front-End callback sequence and pass `True` for the `logoutOnly` argument.
- When the LoginLogout callback completes and you receive the `UIMsg_EndExecution` event, call the `Engine.ShutDown` method again but pass `True` for the **final** parameter.

The second pass ends when you receive the `UIMsg_ShutDownComplete` event. When you receive this event and you are shutting down in preparation to exit the application, the application can safely exit. If you are shutting down because the current user logged out, you can resume normal operation.



Note For a LabVIEW user interface, if the TestStand Engine attempts to launch a dialog box during shutdown and the main application window no longer exists, the user interface hangs. To prevent this hang, ensure that the

engine is closed before you close the front panel of the user interface.

Related concepts:

- [Starting and Shutting Down TestStand from a .NET User Interface](#)
- [User Interface Messages \(UIMessages\)](#)
- [Log in/Log out Execution](#)

Enforcing User Privileges

Most of the dialog boxes you can launch using methods in the TestStand Engine verify user privileges for you and disable controls the user does not have permission to access. However, most of the TestStand API does not enforce privileges to avoid overly restrictive situations. For example, if the TestStand API strictly enforced the execution privilege, the application could not run the LoginLogout callback sequence to allow a user to log in.

The application developer is responsible for enforcing user privileges. TestStand provides a predefined set of user privileges and an Engine method you can call to verify whether the current user has a specific privilege. To determine whether to disable controls or menu items in the application, call the `Engine.CurrentUserHasPrivilege` method. Pass the value of one of the `UserPrivileges` string constants for the **privilegeName** parameter. If no user is currently logged in, the `Engine.CurrentUserHasPrivilege` method always returns `False`. If privilege checking is disabled, the `Engine.CurrentUserHasPrivilege` method always returns `True`.

Related concepts:

- [Privileges](#)

Resource Strings and Internationalization

To provide a way for both the application developer and the end-user to internationalize the text of applications and dialog boxes, TestStand provides a resource string file mechanism for storing all the application text in a central location. You can change the strings the application displays by editing or creating `.ini` files in

a text editor.

To add string support for the internationalization of the application, use the `Engine.GetResourceString` method to obtain the text for the windows and controls. You can add strings to an existing string file or create a new string file in the appropriate directory under the `<TestStand Public>\Components\Language` directory. Ensure that you create a unique category for the strings.

Related concepts:

- [Creating String Resource Files](#)
- [TestStand Directory Structure](#)

Thread Safety of the PropertyObject API and TestStand Variables

You can modify and access the structure, values, and other settings of TestStand variables concurrently from multiple threads without causing crashes, various errors, or data inconsistencies. Use the `PropertyObject.GetSubProperties` and `PropertyObject.GetPropertyObjectElements` methods to obtain references to all subproperties or elements of a property in one atomic call to avoid issues with invalid indexes that result when another thread removes or inserts properties or elements while a different thread iterates through the subproperties or elements. Use synchronization primitives, such as locks or semaphores, if you need to maintain atomicity or control editing and data access across multiple edits or `PropertyObject` API calls.

Actions That Are Thread-Safe in TestStand

The following actions are thread-safe in TestStand:

- Using the `PropertyObject` API or TestStand expressions to access or modify variables from one thread while modifying the structure of such variables in another thread. You can access or modify the parent, the name, and other items that are accessible through the `PropertyObject` API.
- Serializing variables by using the `Engine.SerializeObjects` method with the `SerializationOption_UseBinary` or `SerializationOption_UseXml` options or by

saving a `PropertyObjectFile` in the XML or Binary format while modifying the structure of such variables in another thread. TestStand maintains the serialized data in a consistent state. However, the timing of the serialization code in regard to the concurrent modifications determines what TestStand actually serializes.

Actions That Are Not Thread-Safe in TestStand

The following actions are not thread-safe in TestStand:

- Modifying a type definition in one thread while other threads access the same type definition or its instances.
- Serializing variables using an INI-based format while modifying the structure of such variables in another thread.
- Accessing API properties and methods of a `SequenceContext` object from a thread other than the thread the object belongs to while the thread the `SequenceContext` object belongs to runs the TestStand sequence to which the `SequenceContext` object applies.
- Modifying the structure of built-in properties for steps, sequences, and `SequenceContext` objects while another thread accesses those objects.