TestStand™

Reference Manual

July 2003 Edition
Part Number 323435A-01

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

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

Worldwide Offices

Australia 1800 300 800, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599, Canada (Calgary) 403 274 9391, Canada (Montreal) 514 288 5722, Canada (Ottawa) 613 233 5949, Canada (Québec) 514 694 8521, Canada (Toronto) 905 785 0085, Canada (Vancouver) 514 685 7530, China 86 21 6555 7838, Czech Republic 420 2 2423 5774, Denmark 45 45 76 26 00, Finland 385 0 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, Greece 30 2 10 42 96 427, India 91 80 51190000, Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400, Malaysia 603 9131 0918, Mexico 001 800 010 0793, Netherlands 31 0 348 433 466, New Zealand 1800 300 800, Norway 47 0 66 90 76 60, Poland 48 0 22 3390 150, Portugal 351 210 311 210, Russia 7 095 238 7139, Singapore 65 6226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 2 2528 7227, Thailand 662 992 7519, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on the documentation, send email to techpubs@ni.com.

Important Information

Warranty

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

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

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

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

Copyright

Trademarks

CVI™, IVI™, LabVIEW™, Measurement Studio™, National Instruments™, NI™, NI Developer Zone™, ni.com™, and TestStand™ are trademarks of National Instruments Corporation.

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

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or `ni.com/patents`.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

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

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

Contents

About This Manual

Chapter 1
TestStand Architecture

Chapter 2
Sequence Files and Workspaces

Chapter 3
Executions

Chapter 4
Built-In Step Types

Chapter 5
Module Adapters

Chapter 6
Database Logging and Report Generation

Chapter 7
User Management

Chapter 8
Customizing and Configuring TestStand

Chapter 9
Creating Custom Operator Interfaces

Chapter 10
Customizing Process Models and Callbacks

Chapter 11
Type Concepts

Chapter 12
Standard and Custom Data Types

Chapter 13
Creating Custom Step Types

Chapter 14
Deploying TestStand Systems

Appendix A
Process Model Architecture

Appendix B
Synchronization Step Types

About This Manual

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

Conventions

The following conventions appear in this manual

<> — Angle brackets that contain numbers separated by an ellipsis represent a range of values associated with a bit or signal name—for example, DIO<3..0>.

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

♦ — The ♦ symbol indicates that the following text applies only to a specific product, a specific operating system, or a specific software version.

 — This icon denotes a tip, which alerts you to advisory information.

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

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

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

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

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

1

TestStand Architecture

This chapter describes the National Instruments TestStand architecture and provides an overview of important TestStand concepts and components. It is useful to read *Using TestStand* and the *TestStand System and Architecture Overview Card* before reading this manual.

National Instruments also recommends that you become familiar with the concepts of this chapter before proceeding through this manual.

General Test Executive Concepts

A test executive is a program in which you organize and execute sequences of reusable code modules. Ideally, a test executive allows you to create the modules in a variety of programming environments.

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

- *Code module*—A program module, such as a Windows dynamic link library (`.dll`) or National Instruments LabVIEW VI (`.vi`), containing one or more functions that perform a specific test or other action.
- *Step*—An individual element of a test sequence. A step may 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 is executed can depend on the results of previous steps.
- *Subsequence*—A sequence that another sequence calls. You specify a subsequence call as a step in the calling sequence.
- *Sequence file*—A file that contains the definition of one or more sequences.
- *Sequence editor*—A program that provides a graphical user interface (GUI) for creating, editing, and debugging sequences.

- *Operator interface*—A program that provides a GUI for executing sequences on a production station. A sequence editor and operator interface can be separate application programs or different aspects of the same program.
- *Test executive engine*—A module or set of modules that provide an *application programming interface* (*API*) for creating, editing, executing, and debugging sequences. A sequence editor or operator interface uses the services of a test executive engine.
- *Application Development Environment* (*ADE*)—A programming environment such as LabVIEW, National Instruments LabWindows™/CVI™, or Microsoft Visual Studio .NET, in which you create code modules and operator interfaces.
- *Unit Under Test* (*UUT*)—The device or component you are testing.

Major Software Components of TestStand

This section provides an overview of the major software components of TestStand. For a visual representation of how these components interact, refer to the *TestStand System and Architecture Overview Card*, which is included in your TestStand package. You can also refer to the *TestStand Help* for more information about each of these components.

Note If you are opening help files from the `<TestStand>\Doc\Help` directory, National Instruments recommends that you open `TSHelp.chm`. This file is a collection of all of the *TestStand Help* files and provides a complete table of contents and index.

TestStand Sequence Editor

The TestStand Sequence Editor is an application program in which you create, edit, execute, and debug sequences. The sequence editor gives you access to all TestStand features, such as step types and process models, and features the debugging tools you are familiar with in ADEs such as LabVIEW, LabWindows/CVI (ANSI), and Microsoft Visual Studio .NET. These debugging tools include setting breakpoints; stepping into, out of, or over steps; tracing through program executions; displaying variables; and monitoring variables and expressions during executions.

The TestStand Sequence Editor allows you to start multiple concurrent executions—you can execute multiple instances of the same sequence, or you can execute different sequences at the same time. Each execution instance has its own Execution window. In trace mode, the Execution window displays the steps in the currently executing sequence. If the

execution is suspended, the Execution window displays the next step to execute and provides debugging options.

TestStand Operator Interfaces

TestStand includes several operator interfaces, each of which is a separate application program. These interfaces, which are developed in LabVIEW, LabWindows/CVI, Microsoft Visual Basic .NET, C#, and C++ (MFC), are available in both source and executable formats.

The TestStand Operator Interfaces are fully customizable. Like the TestStand Sequence Editor, the operator interfaces allow you to start multiple concurrent executions, set breakpoints, and single-step. However, the operator interfaces do not allow you to modify sequences, and they do not display sequence variables, sequence parameters, step properties, and so on.

You can use the source code of the operator interfaces as a starting point for customization or as a model for your own operator interface. Refer to Chapter 9, *Creating Custom Operator Interfaces*, for more information about the operator interfaces that are included in TestStand.

TestStand User Interface Controls

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

Refer to the *TestStand Help* and to Chapter 9, *Creating Custom Operator Interfaces*, for more information about the TestStand UI Controls. You can also refer to the *TestStand User Interface Controls Reference Poster*, which is included in your TestStand package, for an illustrated overview of the controls and API.

TestStand Engine

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

For more information about the TestStand API, refer to the *TestStand Help*.

Module Adapters

Most steps in a TestStand sequence invoke code in another sequence or in a code module. When invoking code in a code module, TestStand must know the type of code module, how to call it, and how to pass parameters to it. The different types of code modules include LabVIEW VIs; C functions in source, object, or library modules created in LabWindows/CVI; C/C++ functions in DLLs; objects in .NET assemblies; objects in ActiveX automation servers; and subroutines in HTBasic. TestStand must also know the list of parameters required by the code module. TestStand uses *module adapters* to obtain this knowledge.

TestStand includes the following module adapters:

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

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

Refer to Chapter 5, *Module Adapters*, for more information about the module adapters included in TestStand.

TestStand Building Blocks

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

Variables and Properties

TestStand stores data values in *variables* and *properties*. Variables are properties you can freely create in certain contexts. You can have variables that are *global* to a sequence file or local to a particular sequence. You can also have *station global variables*, which have values that are persistent across different executions and even across different invocations of the sequence editor or operator interfaces. The TestStand Engine maintains the value of station global variables in a file on the computer on which it is installed.

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

Each step in a sequence can have properties. For example, a step might have a floating point measurement code property. A step's type determines its set of properties. Refer to the *Step Types* section of this chapter for more information about types of steps.

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

Expressions

In TestStand, you can use the values of variables and properties in numerous ways, such as passing a variable to a code module or using a property value to determine whether to execute a step. For these same purposes, you may also want to 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 in the sequence context that are active when TestStand evaluates the expression.

The following is an example of an expression:

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

You can use an expression wherever you would use a simple variable or property value. TestStand supports all applicable expression operators and syntax that you would use in C, C++, Java, and Visual Basic .NET.

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

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

Categories of Properties

A property is a storage space for information. A property can store a single value or an array of values of the same data type. Each property has a name.

A value can be a number, string, Boolean, .NET object reference, or an 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. Arrays of values can have multiple dimensions.

The following major categories of properties are defined according to the kinds of values they contain:

- *Single-valued property*—Contains a single value. The four types of single-valued properties—number, string, Boolean, and object reference—correspond to the four value types supported by TestStand.
- *Array property*—Contains an array of values. TestStand supports the following array properties: number, string, Boolean, and object reference.
- *Property-array property*—Contains a value that is an array of subproperties of a single type.
- *Container property*—Contains no values and can contain multiple subproperties. Container properties are analogous to structures in C/C++ and to clusters in LabVIEW.

Standard and Custom Data Types

When you create a variable or property, you specify its data type. In some cases, you use a simple data type such as a number or a Boolean. In other cases, you may want to define your own data type by adding subproperties to a container to create an arbitrarily complex data structure. Define your own data type by creating a *named data type*. When you create a named data type, you can reuse it with variables or properties. Although each variable or property you create with a named data type has the same data structure, they can contain different values.

When you create a variable or property, you can use both built-in property types and the named data types.

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

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

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

Built-In and Custom Properties

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

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

Steps

A sequence consists of a series of steps. In TestStand, a step can perform many actions, such as initializing an instrument, performing a complex test, or making a decision that affects 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 also have custom properties. For steps that call code modules, custom step properties are useful for storing parameters to pass to the code module for the step. They also serve as a place for the code module to store its results. You can also use the TestStand API to access the values of custom step properties from within code modules.

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

Step Types

Just as each variable or property has a data type, each step has a *step type*. A step type can contain any number of custom properties. Each step of that type includes the custom step properties in addition to the built-in step properties. While 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. TestStand includes a number of predefined step types. For a description of these step types, refer to Chapter 4, *Built-In Step Types*.

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

Source Code Templates

TestStand also allows you to define *source code templates* for new step types. When you create a new step of a particular type, you can use a source code template to generate source code for the step's code module. You can specify different source code templates for the different module adapters.

Sequences

A TestStand sequence consists of the following components:

- A group of setup steps (Setup step group)
- A main group of steps (Main step group)
- A group of cleanup steps (Cleanup step group)
- Sequence local variables
- Parameters
- Built-in sequence properties

Step Groups

A sequence contains the following groups of steps: Setup, Main, and Cleanup. 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 Unit Under Test (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.

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

If a run-time error occurs in a sequence, TestStand reports the run-time error to the calling sequence. Execution in the calling sequence then jumps to the Cleanup step group in that 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 of all the sequences in the sequence call stack.

Sequence Local Variables

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

Adapter. You can also access local variables from code modules of steps in the sequence using the TestStand API.

Note TestStand can only pass data to LabVIEW VIs by value. LabVIEW does not support passing data by reference.

Sequence Parameters

Each sequence has its own list of parameters. Use these parameters to pass data to a sequence when you call that sequence as a subsequence. Using parameters in this way is analogous to passing arguments to a function call or wiring data to terminals when you call a subVI in LabVIEW. 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 either select a value to pass to the parameter, or use the default value specified by the parameter. You can pass sequence parameters by value or by reference to any step in the sequence that calls a subsequence or any step that calls a code module that uses the LabVIEW, LabWindows/CVI, C/C++ DLL, .NET, or ActiveX/COM Adapter.

You can also access parameters from code modules of steps in the sequence by using the TestStand API.

Note TestStand can only pass data to LabVIEW VIs by value. LabVIEW does not support passing data by reference.

Built-in Sequence Properties

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

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

Sequence Files

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

Sequence files have built-in properties that you can specify using the Sequence File Properties dialog box. For example, you can specify Load

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

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

Process Models

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

Some commercial test executives implement their process model internally and do not allow you to modify them. Other test executives do not come with a process model at all. TestStand comes with three default process models that you can modify or replace: the Sequential model, the Batch model, and the Parallel model. You can use the Sequential model to run a test sequence on one UUT at a time. The Parallel and Batch models allow you to run the same test sequence on multiple UUTs at the same time.

TestStand provides a mechanism for defining your own process model, which is a sequence file that enables you to write different test sequences without repeating standard testing operations in each sequence. This modification is essential since the testing process can vary according to your production line, your production site, or the systems and practices of your company. You can edit a process model in the same way that you edit your other sequence files.

Station Model

You can specify a process model file to use for all sequence files. This process model file is called the *station model*. The Sequential model is the default station model file. You can use the Station Options dialog box to select a different station model, or to allow individual sequence files to specify their own process model file.

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

Main Sequence and Client Sequence File

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

You begin an execution from a Main sequence in one of your sequence files. TestStand determines which process model file to use with the Main sequence. TestStand uses the station model file unless the sequence file specifies a different process model file and you have set the Allow Other Models option in the Station Options dialog box to allow sequence files to override your station model setting.

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

Entry Points

A process model defines a set of *entry points*. Each entry point is a sequence in the process model file. By defining multiple entry points in a process model, you give the test station operator different ways to invoke a Main sequence or configure the process model.

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

Refer to Chapter 3, *Executions*, for more information about entry points.

Automatic Result Collection

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

Each sequence has a local array that stores the results of each step. The contents in the results for each step can vary depending on the step type. When TestStand stores the results for a step into the array, it adds information such as the name of the step and its position in the sequence. For a step that calls a sequence, TestStand also adds the result array from the subsequence.

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

Callback Sequences

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

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

TestStand defines three categories of callbacks: Model callbacks, Engine callbacks, and Front-End callbacks. The categories are based on the entity that invokes the callback and the location in which you define the callback. Model callbacks allow you to customize the behavior of a process model for each Main sequence that uses it. Engine callbacks are defined by the TestStand Engine and are invoked at specific points during execution. Front-End callbacks are called by operator interface programs to allow multiple operator interfaces to share the same behavior for a specific operation.

Table 1-1 illustrates the different types of callbacks.

Table 1-1. Callback Types

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

Sequence Executions

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

2

Sequence Files and Workspaces

This chapter describes TestStand sequence files and workspaces.

Sequence Files

A TestStand sequence file (`.seq`) is a file that contains any number of sequences, a set of types used in the sequence file, and any global variables shared by steps and sequences in the file.

Types of Sequence Files

TestStand contains the following types of sequence files:

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

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

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

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

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

Sequence File Callbacks

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

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

Sequence File Globals

Each sequence file can contain any number of global variables. These variables are accessible from any step or sequence within the sequence file in which they are defined. View and edit the global variables in the Sequence File Globals view.

Refer to the *TestStand Help* for more information about the Sequence File Globals view.

Sequence File Type Definitions

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

Refer to the *TestStand Help* for more information about the Sequence File Types view. Refer to Chapter 11, *Type Concepts*, for more information about types and type editing.

Comparing and Merging Sequence Files

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

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

Sequences

Each sequence can contain steps, parameters, and local variables. View and edit the contents of a specific sequence in the Individual Sequence view.

Sequences have properties that you can view and edit from the Sequence Properties dialog box. For more information about the Sequence Properties dialog box, refer to the *TestStand Help*.

Step Groups

Sequences contain their steps in three groups: Setup, Main, and Cleanup. You can view and edit the step groups on their corresponding tabs in the Sequence view of the Sequence File window.

The Setup step group contains steps that initialize or configure your instruments, fixtures, and UUTs. The Main step group contains steps that test your UUTs. The Cleanup step group contains steps that power down or release handles to your instruments, fixtures, and UUTs.

Refer to the *TestStand Help* for more information about the Step Group tabs.

Parameters

Each sequence has its own list of parameters. Use these parameters to pass data to and from a sequence when you call that sequence as a subsequence. You can view and edit the parameters for a sequence on the Parameters tab in the Sequence view of the Sequence File window.

Refer to the *TestStand Help* for more information about the Parameters tab.

Local Variables

Use local variables to store data relevant to the execution of the sequence. You can access local variables from within steps and code modules. You can also use local variables for maintaining counts, holding intermediate values, or any other purpose. View and edit the local variables for a sequence on the Locals tab in the Sequence view of the Sequence File window.

Refer to the *TestStand Help* for more information about the Locals tab.

Sequence File Window and Views

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

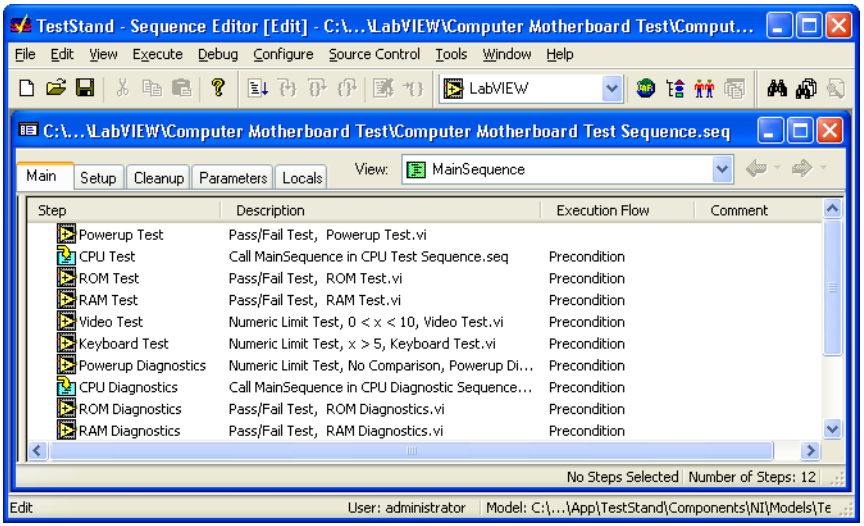


Figure 2-1. Sequence File Window

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

Use the View ring control, located at the top right of the Sequence File window, to select the aspect of the file that you want to display. The View ring control contains the following display options:

- **All Sequences**—Accesses a list of sequences in a file. Use this view to create new sequences and to cut, copy, and paste sequences. You can also drag and drop sequences from this view to the All Sequences view in another Sequence File window.
- **Individual Sequence**—Accesses the contents of a specific sequence.
- **Sequence File Globals**—Accesses the global variables for the sequence.
- **Sequence File Types**—Accesses the types that the sequence file contains.

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

Workspaces

A TestStand workspace file (`.tsw`) contains references to any number of TestStand project files. A TestStand project file (`.tpj`) contains references to any number of other files of any type.

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

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

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

Source Code Control

You can use workspace and project files in TestStand to access your files in a source code control (SCC) system. To perform SCC operations on your files from within TestStand, select a SCC provider on the Source Control tab on the Station Options dialog box.

Note National Instruments has tested TestStand with the following source code control providers: Microsoft Visual SourceSafe, Perforce, MKS Source Integrity, and Rational ClearCase.

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

System Deployment

The TestStand Deployment Utility uses workspace and project files to collect all of the files required to successfully distribute your test system to a target computer. The deployment utility also creates an installer for your test system.

Refer to Chapter 14, *Deploying TestStand Systems*, for more information about system deployment and the TestStand Deployment Utility.

3

Executions

This chapter describes TestStand executions.

What is an Execution?

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

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

Note Built-in properties of steps and sequences are flagged to be shared 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, TestStand maintains an *execution pointer* that points to the current step, a call stack, and a run-time copy of the local variables and custom properties for all sequences and steps on the call stack.

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

Sequence Context

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

TestStand also maintains a sequence context that contains references to all global variables and step properties in the active sequence. The contents of a sequence context can vary depending on the currently executing step. For more information about the contents of the sequence context, refer to the *TestStand Help*.

Using the Sequence Context

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

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

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

You can pass a reference to a SequenceContext object from a code module. In code modules, you access the value of a variable or property by using PropertyObject methods in the TestStand API with the sequence context. As with expressions, you must specify a path from the sequence context to the particular property or variable. Refer to Chapter 5, *Module Adapters*, for more information about passing the SequenceContext object to a code module for each adapter. Refer to the *TestStand Help* for more information about accessing the properties in the sequence context from code modules.

Select **View»Browse Sequence Context** in the sequence editor to open a tree view containing the names of variables, properties, and sequence parameters that you can access from expressions and code modules. Refer to the *TestStand Help* for more information about the Sequence Context tree view.

Note Some properties are not populated until run time.

Lifetime of Local Variables, Parameters, and Custom Step Properties

Multiple instances of a sequence can run at the same time. This situation can occur when you call a sequence recursively or when a sequence runs in multiple concurrent threads. Each instance of the sequence has its own copy of the sequence parameters, local variables, and custom properties of each step. When a sequence completes, TestStand discards the values of the parameters, local variables, and custom properties.

Sequence Editor Execution Window

The sequence editor displays each execution in a separate Execution window. Figure 3-1 illustrates the Execution window.

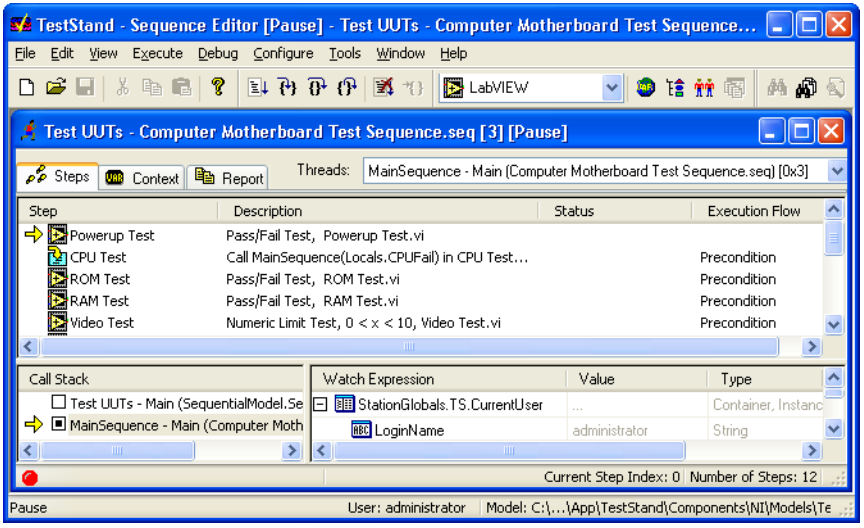


Figure 3-1. Sequence Editor Execution Window

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

Starting an Execution

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

Execution Entry Points

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

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 are actually running an entry point sequence in a process model file. The Execution entry point sequence, in turn, invokes the Main sequence one time or multiple times.

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

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

Executing a Sequence Directly

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

Tip Executing a sequence directly is best for performing unit testing or debugging.

Interactively Executing Steps

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

There are two ways that you can run steps in interactive mode:

- Run steps interactively from a Sequence File window. This creates a new execution called a *root interactive execution*. You can set station options to control whether the Setup and Cleanup step groups of the sequence run as part of a root interactive execution.
- Run steps interactively from an existing Execution window for a normal execution that is suspended at a breakpoint by selecting Run Selected Steps or Loop On Selected Steps. The selected steps run within the context of the normal execution. This is called a *nested interactive execution*.

  The steps that you run interactively can access the variable values of the normal execution and add to its results. If you used the process model for the original execution, these results are included in the test report. When the selected steps complete, the execution returns to the step at which it was originally suspended.

In interactive mode, the selected steps run in the order in which they appear in the sequence. A configurable station option specifies whether a branch operation is allowed to a specific step or a non-selected step, or whether only the selected steps in a sequence execute, regardless of any branching logic that the sequence contains.

To configure whether TestStand evaluates preconditions when executing interactively, select **Configure»Station Options** and enable the **Evaluate Preconditions** option in the Interactive Executions section of the **Execution** tab on the Station Options dialog box. You can also use this dialog box to configure whether interactive executions branch to unselected steps in the Branching Mode control.

Terminating and Aborting Executions

The menus in the sequence editor and operator interfaces include commands that allow you to stop execution before the execution has completed normally. The TestStand API has corresponding methods that allow you to stop execution from inside of a code module or determine whether the execution has been stopped. You can stop one execution or all executions by issuing a stop request at any time. Stop requests do not take

effect in each execution until the currently executing code module returns control.

You can stop executions in two ways:

- When you *terminate* an execution, all the Cleanup step groups in the sequences on the call stack run before execution ends. Also, if you terminate an execution while the client sequence file is still running, the process model will continue to run, possibly testing the next UUT or generating a test report.
- When you *abort* an execution, the Cleanup step groups do not run, and the process model will 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 that the Cleanup step groups can return your system to a known state.

Tip You should only abort an execution when you are debugging and when you are sure that it is safe to skip the cleanup steps for a sequence.

Result Collection

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

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

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

Through the TestStand API, a code module 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.

Custom Result Properties

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

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

Custom Step Property	Step Types that Use the Property
Error.Code	All
Error.Msg	All
Error.Occurred	All
Status	All
Common	All
Numeric	Numeric Limit Test, Multiple Numeric Limit Test
PassFail	Pass Fail Test
String	String Value Test
ButtonHit	Message Popup
Response	Message Popup
ExitCode	Call Executable
NumPropertiesRead	Property Loader
NumPropertiesApplied	Property Loader
ReportText	All
Limits.Low	Numeric Limit Test, Multiple Numeric Limit Test
Limits.High	Numeric Limit Test Multiple Numeric Limit Test

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

Custom Step Property	Step Types that Use the Property
Comp	String Value Test, Numeric Limit Test, Multiple Numeric Limit Test
Measurement	Multiple Numeric Limit Test
AsyncID	Sequence Call
AsyncMode	Sequence Call

Note Table 3-1 does not include the result properties for Synchronization, IVI, or Database step types. For more information about these step types, refer to Appendix B, *Synchronization Step Types*; Appendix C, *IVI Step Types*; and Appendix D, *Database Step Types*.

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

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

The Common result subproperty uses the CommonResults custom data type. The Common property is a subproperty of the Result property for every built-in step type. Consequently, you can add a subproperty to the result of every step type by adding a subproperty to the definition of the CommonResults custom data type.

Be aware that if you modify CommonResults without incrementing the type version number, you may see a type conflict when you open other sequence files. These conflicts can include `FrontEndCallbacks.seq` when you are logging in or out. TestStand will automatically prompt you to increment the version number when saving changes to any data type or step type.

Standard Result Properties

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

Table 3-2. Standard Step Result Properties

Standard Result Property	Description
TS.StartTime	Time at which the step began executing, specifically, the number of seconds since the TestStand Engine initialized.
TS.TotalTime	Number of seconds the step took to execute. This time includes the time for all step options including preconditions, expressions, post actions, module loading, and module execution.
TS.ModuleTime	Number of seconds that 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.
TS.Id	A number that TestStand assigns to the step result. The number is unique with respect to all other step results in the current TestStand session.
TS.InteractiveExeNum	A number that TestStand assigns to an interactive execution. The number is unique with respect to all other interactive executions in the current TestStand session. TestStand only adds this property if you run the step interactively.
TS.StepType	Name of the step type.
TS.Server	This property contains the name of the server machine on which the step runs the subsequence it calls. This result property only exists for Sequence Call steps that run subsequences on a remote machine.
TS.StepCausedSequenceFailure	This property only exists if the step fails. The value is `True` if the step failure causes the sequence to fail. The value is `False` if the step failure does not cause the sequence to fail or if the sequence has already failed.

Subsequence Results

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

Table 3-3. Property Names for Subsequence Results

Result Subproperty Name	Type of Subsequence Call
TS.SequenceCall	Sequence Call
TS.PostAction	Post Action Callback
TS.SequenceFilePreStep	SequenceFilePreStep Callback
TS.SequenceFilePostStep	SequenceFilePostStep Callback
TS.ProcessModelPreStep	ProcessModelPreStep Callback
TS.ProcessModelPostStep	ProcessModelPostStep Callback
TS.StationPreStep	StationPreStep Callback
TS.StationPostStep	StationPostStep Callback
TS.SequenceFilePreInteractive	SequenceFilePreInteractive Callback
TS.SequenceFilePostInteractive	SequenceFilePostInteractive Callback
TS.ProcessModelPreInteractive	ProcessModelPreInteractive Callback
TS.ProcessModelPostInteractive	ProcessModelPostInteractive Callback
TS.StationPreInteractive	StationPreInteractive Callback
TS.StationPostInteractive	StationPostInteractive Callback
TS.SequenceFilePostResultListEntry	SequenceFilePostResultListEntry Callback
TS.ProcessModelPostResultListEntry	ProcessModelPostResultListEntry Callback
TS.StationPostResultListEntry	StationPostResultListEntry Callback
TS.SequenceFilePostStepRuntimeError	SequenceFilePostStepRuntimeError Callback
TS.ProcessModelPostStepRuntimeError	ProcessModelPostStepRuntimeError Callback
TS.StationPostStepRuntimeError	StationPostStepRuntimeError Callback
TS.SequenceFilePostStepFailure	SequenceFilePostFailure Callback

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

Result Subproperty Name	Type of Subsequence Call
TS.ProcessModelPostStepFailure	ProcessModelPostFailure Callback
TS.StationPostStepFailure	StationFilePostFailure Callback

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

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

Loop Results

When you configure a step to loop, you can use the Record Result of Each Iteration option on the Loop Options tab on the Step Properties dialog box 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 come immediately after the result for the step as a whole.

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

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

Report Generation

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

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

Engine Callbacks

TestStand specifies a set of callback sequences that it invokes at specific points during execution. These callbacks are called Engine callbacks.

Engine callbacks are a way for you to tell TestStand to call certain sequences before and after the execution of individual steps, before and after interactive executions, after loading a sequence file, and before unloading a sequence file. Because the TestStand Engine controls the execution of steps and the loading and unloading of sequence files, TestStand defines the set of Engine callbacks and their names.

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

Step Execution

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

Table 3-4. Order of Actions that a Step Performs

Action Number	Description	Remarks
1	Enter batch synchronization section	If option is set
2	Acquire step lock	If option is set
3	Allocate step result	—

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

Action Number	Description	Remarks
4	Evaluate precondition	If `False`, go to Action Number 27
5	Check run mode	—
6	Load module if not already loaded	—
7	Evaluate Loop Initialization expression	Only if looping
8	Evaluate Loop While expression, skip to Action Number 22 if `False`	Only if looping
9	Allocate loop iteration result	Only if looping
10	Call Pre-Step Engine callbacks	—
11	Evaluate Pre-Expression	—
12	Call Pre-Step substeps for step type	—
13	Call module	—
14	Call Post-Step substeps for step type	TestStand calls Post-Step substeps even if the user code module generates a run-time error. This enables Post-Step substeps to perform error handling, if appropriate.
15	Evaluate Post-Expression	—
16	Evaluate Status expression	—
17	Call Post-Step Engine callbacks	—
18	Call PostStepFailure Engine callback	Only if loop iteration fails
19	Fill out loop iteration result	Only if looping
20	Call PostResultListEntry Engine callback	Only if looping

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

Action Number	Description	Remarks
21	Evaluate Loop Increment expression, return to Action Number 8	Only if looping
22	Evaluate Loop Status expression	Only if looping
23	Unload module if required	—
24	Update sequence failed state	
25	Call PostStepFailure Engine callback	Only if step fails
26	Execute post action	—
27	Fill out step result	—
28	Call PostResultListEntry Engine callback	—
29	Release step lock	If option is set
30	Exit batch synchronization section	If option is set

Usually, a step performs only a subset of these actions, depending on the configuration of the step and the test station. When TestStand detects a run-time error, it calls the PostStepRuntimeError callback. If these callbacks are not defined or if they do not reset the error state for the step, TestStand proceeds to Action Number 27. If a run-time error occurs in a loop iteration, TestStand performs Action Number 19 before performing Action Number 27.

Step Status

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

Table 3-5. Standard Values for the Status Property

String Value	Meaning	Source of the Status Value
`Passed`	Indicates that the step performed a test that passed.	Step or code module
`Failed`	Indicates that the step performed a test that failed.	Step or code module
`Error`	Indicates that a run-time error occurred.	TestStand
`Done`	Indicates that the step completed without setting its status.	TestStand
`Terminated`	Indicates that the step called a subsequence in which execution terminated. Only occurs for Sequence Call steps for which you enable the Ignore Termination option on the Run Options tab on the Step Properties dialog box.	TestStand
`Skipped`	Indicates that the step did not execute because the run mode for the step is Skip.	TestStand
`Running`	Indicates that the step is currently running.	TestStand
`Looping`	Indicates that the step is currently running in loop mode.	TestStand

Failures

TestStand considers a step to have failed if the step executes and the step status is `Failed`. If you enable the Step Failure Causes Sequence Failure option on the Run Options tab on the Step Properties dialog box, TestStand sets the sequence status to `Failed` when the 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.

Note For most step types, the Step Failure Causes Sequence Failure option is enabled by default.

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, this option is disabled.

Run-Time Errors

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

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

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

4

Built-In Step Types

This chapter describes the core set of built-in step types that TestStand provides and groups them into the following categories:

- Step types that 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 and Statement perform an action that you configure in an editing dialog box that is specific to the step type.

Note TestStand also includes sets of application-specific step types. For example, TestStand provides sets of step types that make it easier to synchronize multiple threads, control IVI instruments, and access databases. For more information about these step types, refer to Appendix B, *Synchronization Step Types*; Appendix C, *IVI Step Types*; and Appendix D, *Database Step Types*.

Overview

This section describes general features of built-in step types.

Using Step Types

Use step types when you insert steps in the Setup, Main, and Cleanup tabs of an individual sequence view in the Sequence File window. The Insert Step item in the context menu displays a submenu that shows all of the available step types.

Figure 4-1 shows the submenu for the Insert Step item.

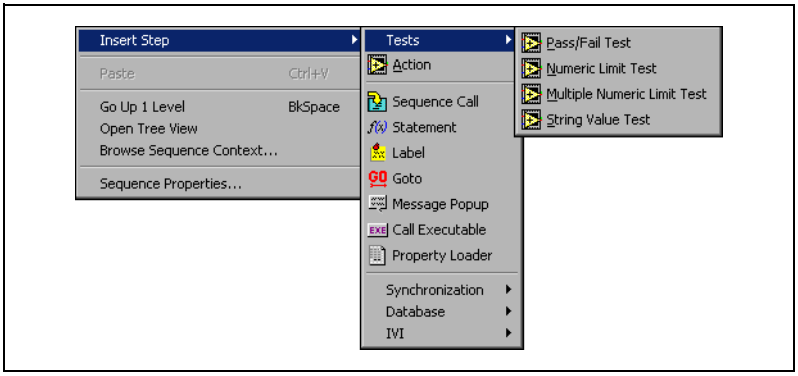


Figure 4-1. Insert Step Submenu

When you select a step type in the submenu, TestStand creates a step using the step type and module adapter indicated in the submenu entry. After you insert the step, select **Specify Module** from the context menu to specify the code module or sequence, if any, that the step calls. The Specify Module command launches a dialog box that is different for each adapter. The generic name for the dialog box is the Specify Module dialog box. Refer to Chapter 5, *Module Adapters*, and the *TestStand Help* for information about the Specify Module dialog box for each adapter.

For each step type, other items can appear in the context menu above the Specify Module item. For example, the Edit Limits item appears in the context menu for Numeric Limit Test steps, and the Edit Pass/Fail Source item appears in the context menu for Pass/Fail Test steps. Select the menu item to launch a *step-type-specific dialog box,* in which you can modify step properties that are specific to the step type. Refer to the *TestStand Help* for information about the menu item for each of the built-in step types.

To modify step properties that are common to all step types, select **Properties** from the context menu, double-click the step, or press **<Enter>** with the step selected. The Step Properties dialog box contains buttons to open the Specify Module dialog box and the step-type-specific dialog boxes. Refer to the *TestStand Help* for more information about the Step Properties dialog box.

Built-In Step Properties

TestStand steps feature a number of built-in properties which you can specify using the various options on the Step Properties dialog box. The following list explains the capabilities of each built-in step property:

General Tab

- **Preconditions**—Set this property to specify the conditions that must be `True` for TestStand to execute the step during the normal flow of execution in a sequence.

Run Options Tab

- **Load/Unload Options**—Set this property to control when TestStand loads and unloads the code modules or subsequences that are invoked by each step.
- **Run Mode**—Set this property to specify whether TestStand skips a step or forces the step to pass or fail without executing the step's code module.
- **Record Results**—Set this property to specify whether TestStand collects the results for this step. Refer to the *Result Collection* section of Chapter 3, *Executions*, for more information.
- **Step Failure Causes Sequence Failure**—Set this property to specify whether TestStand sets the status of the sequence to `Failed` when the status of the step is `Failed`.
- **Ignore Run-Time Errors**—Set this property to specify whether TestStand continues execution normally after the step even though a run-time error occurs in the step.

Post Actions Tab

- **Post Actions**—Set this property to execute other sequences or jump to other steps after executing the step, depending on the pass/fail status of the step or any custom condition.

Loop Options Tab

- **Loop**—Set this property to cause a single step to execute multiple times before executing the next step. You can specify the conditions under which to terminate the loop. You can also specify whether to collect results for each loop iteration, for the loop as a whole, or for both.

Expressions Tab

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

Synchronization Tab

- **Synchronization**—Set this property to specify whether a step should block another instance of the step from executing at the same time in a different thread.

Switching Tab

- **Switching**—Set this property to specify whether TestStand performs any switching operations when the step executes.

Use the Step Properties dialog box to modify the values of the built-in step properties. You can usually modify the values of custom step properties using a dialog box specific to the step type. If the step type does not have a dialog box for the custom properties, select **View Contents** from the step's context menu to view the custom properties for that step. Although code modules usually do not modify the values of the built-in step properties at run time, they often modify and read the values of the custom step properties when determining the pass/fail status.

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

Custom Properties That Are Common to All Built-In Step Types

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

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

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

- **Step.Result.Error.Msg**—Message string that describes the error that occurred.
- **Step.Result.Status**—Specifies the status of the last execution of the step, such as `Done`, `Passed`, `Failed`, `Skipped`, or `Error`. TestStand documentation refers to this property as the *step status*.
- **Step.Result.Common**—Placeholder container that you can customize. To customize the container, modify the CommonResults standard data type. Refer to the *Using Data Types* section of Chapter 12, *Standard and Custom Data Types*, for more information about standard TestStand data types.
- **Step.Result.ReportText**—Contains a message string that TestStand includes in the report.

Refer to Chapter 5, *Module Adapters*, for more information about the property assignments that module adapters automatically perform to and from step properties.

Step Status, Error Occurred Flag, and Run-Time Errors

The error occurred flag can become `True` in the following situations:

- A run-time error condition occurs, and the code module or module adapter sets the value to `True`.
- An 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 as follows:

- Makes no evaluation of status and post-expressions for a step. Instead, TestStand sets the step status to `Error`.
- Evaluates the Ignore Run-Time Errors step property.
  - If `False`, TestStand reports the run-time error to the sequence.
  - If `True`, TestStand continues execution normally after the step.

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`, the TestStand Engine responds as follows: when the step status is still `Looping` or `Running`, TestStand changes the step status to `Done`.

The step status becomes `Passed` or `Failed` only when a code module, module adapter, or step type explicitly sets the step status to `Passed` or `Failed`.

Refer to Chapter 5, *Module Adapters*, for more information about the assignments that module adapters make to and from step properties.

Step Types That You Can Use with Any Module Adapter

TestStand comes with five built-in step types that you can use with any module adapter: Pass/Fail Test, Numeric Limit Test, Multiple Numeric Limit Test, String Value Test, and Action. When you insert a step in a sequence, you must select a module adapter from the Adapter ring control, which is located on the sequence editor toolbar. TestStand then assigns the adapter you selected to that step.

The icon for the adapter appears as the icon for the step. The icons for the different adapters are as follows:


LabVIEW Adapter


LabWindows/CVI Adapter


C/C++ DLL Adapter


.NET Adapter


ActiveX/COM Adapter


HTBasic Adapter


Sequence Adapter


<None>

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

Note Once you add a step, you can change the adapter associated with that step in the Step Properties dialog box for the step.

To specify the code module that the step calls, select **Specify Module** from the step context menu or click **Specify Module** on the **General** tab on the Step Properties dialog box. Each step has a Specify Module dialog box that corresponds to its module adapter. Refer to the *TestStand Help* for more information about the Specify Module dialog box for each module adapter.

Action

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

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

Pass/Fail Test

Use a Pass/Fail Test step to call a code module that makes its own pass/fail determination.

After the code module executes, the Pass/Fail Test step type evaluates the Step.Result.PassFail property. If Step.Result.PassFail is `True`, the step type sets the step status to `Passed`. If Step.Result.PassFail is `False`, the step type sets the step status to `Failed`.

The following are the different ways that a code module can set the value of Step.Result.PassFail:

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

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

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

By default, the step type uses the value of the Step.Result.PassFail Boolean property to determine whether the step passes or fails. To customize the Boolean expression that determines whether the step passes, select **Edit Pass/Fail Source** from the context menu for the step or click **Edit Pass/Fail Source** on the Step Properties dialog box.

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

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

  This property exists to maintain compatibility with previous test executives. Usually, code modules you develop for TestStand receive data as input parameters or access data as properties using the TestStand API.
- **Step.DataSource**—Specifies the Boolean expression that the step uses to set the value of Step.Result.PassFail. The default value of the expression is `"Step.Result.PassFail"`, which has the effect of using the value that the code module sets.

  You can customize this expression if you do not want to set the value of Step.Result.PassFail in the code module. For example, you can set the data source expression to refer to multiple variables and properties, such as, `RunState.PreviousStep.Result.Numeric * Locals.Attenuation > 12`.

Numeric Limit Test

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

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

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

  Refer to *Using LabVIEW with TestStand* and *Using LabWindows/CVI with TestStand* for more information about the assignments that the module adapters automatically make to and from step properties for legacy code modules in LabVIEW and LabWindows/CVI.
- **All Adapters**—Use the TestStand API to set the value of Step.Result.Numeric directly in a code module.

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

The Numeric Limit Test step type defines the following step properties in addition to the common custom properties:

- **Step.Result.Numeric**—Specifies the numeric measurement value. Usually, you set this value in the code module.
- **Step.Limits.High** and **Step.Limits.Low**—Specify the limits for the comparison expression.
- **Step.Comp**—Specifies the type of comparison, such as `EQ`.
- **Step.Result.Units**—Specifies a label that indicates the unit of measurement.

- **Step.InBuf**—Specifies an arbitrary string that the LabVIEW and LabWindows/CVI Adapters pass to the test in the **Input Buffer** control or **tTestData** structure of legacy code modules.

  This property exists to maintain compatibility with previous test executives. Usually, code modules you develop for TestStand receive data as input parameters or access data as properties using the TestStand API.
- **Step.DataSource**—Specifies a numeric expression that the step type uses to set the value of Step.Result.Numeric. The default value of the expression is `"Step.Result.Numeric"`, which has the effect of using the value that the code module sets. You can customize this expression if you do not want to set the value of Step.Result.Numeric in the code module.

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

For more information about the Edit Numeric Limit Test dialog box, refer to the *TestStand Help*.

Multiple Numeric Limit Test

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

The Multiple Numeric Limit Test allows you to test limits for any number of measurements. Each measurement can have independent limits, units, display formats, data sources, and comparison types. A Multiple Numeric Limit Test step passes if all of its measurements pass. Configure each measurement the same way you configure an individual Numeric Limit Test step. Refer to the *TestStand Help* for more information about the Multiple Numeric Limit Test step and the Edit Multiple Numeric Limit Test dialog box.

The Multiple Numeric Limit Test step type defines the following step properties in addition to the common custom properties:

- **Step.Result.Measurement**—An array that stores the measurements you configure for the step. Each element of the measurement array is an instance of the NI_LimitMeasurement data type. The NI_LimitMeasurement type defines the following fields:
  - **Limits.High** and **Limits.Low**—Specify the limits to which the step compares the measurement value.
  - **Units**—Specifies a label that describes the measurement units for the limits and the measurement value.
  - **Comp**—Specifies the type of comparison, such as `EQ`.
  - **Data**—Stores the numeric measurement value. The step obtains this value from the corresponding element in Step.NumericArray or from the *data source* you specify.
  - **Status**—Stores the result of the comparison of the measurement value with the limits. The result is either `Passed` or `Failed`.
- **Step.DataSource**—Specifies an expression that identifies the numeric array that provides the data values for all measurements when you do not use a separate data source for each measurement.
- **Step.NumericArray**—Provides a numeric array that is the default data source that Step.DataSource specifies.
- **Step.UseIndividualDataSources**—Specifies whether the step stores separate data source expressions for each measurement in the Step.DataSourceArray. If this property is `False`, the step obtains the data values for each measurement from the numeric array that the Step.DataSource property specifies.
- **Step.DataSourceArray**—Specifies a data source for each measurement element in the measurement array.

String Value Test

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

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

- **LabVIEW Adapter**—Specify `Step.Result.String` as the Value expression for a Numeric output of a VI in the Edit LabVIEW VI Call dialog box.
- **LabWindows/CVI**, **C/C++ DLL**, **.NET**, **ActiveX/COM**, or **Sequence Adapter**—Pass `Step.Result.String` as a reference parameter to a code module.

  LabVIEW or **LabWindows/CVI Adapter**—The LabVIEW and LabWindows/CVI Adapters update the value of Step.Result.String automatically, after calling legacy code modules. The LabVIEW Adapter updates the value of Step.Result.String, based on the value of String Measurement in the **Test Data** cluster that the VI returns. The LabWindows/CVI Adapter updates the value of Step.Result.String, based on the value of the stringMeasurement field of the **tTestData** parameter that it passes to the C function.

  Refer to *Using LabVIEW with TestStand* and *Using LabWindows/CVI with TestStand* for more information about the assignments that the module adapters automatically make to and from step properties for legacy code modules in LabVIEW and LabWindows/CVI.
- **All Adapters**—Use the TestStand API to set the value of Step.Result.String directly in a code module.

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

Refer to the *TestStand Help* for more information about the String Value Test step and the Edit String Value Test dialog box.

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

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

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

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

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

Step Types That Work With a Specific Module Adapter

This section describes step types that work with a specific module adapter.

Sequence Call

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

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

Note You can enable or disable the Step Failure Causes Sequence Failure option on the Run Options tab on the Step Properties dialog box.

Refer to the *TestStand Help* for more information about using the Step Properties dialog box for the Sequence Adapter.

The Sequence Call step type does not define any additional step properties other than the custom properties that are common to all steps.

TestStand adds the following properties to the results for Sequence Call steps that are configured to run the sequence in a new thread or execution. These properties are not subproperties of the Result property for the Sequence Call step type.

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

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

Step Types That Do Not Use Module Adapters

This section describes step types that do not use module adapters. When you create an instance of one of these step types, you only use the Edit <Step Name> dialog box, which you access through the step's context menu, to configure the step. You do not specify a code module.

Statement

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

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

Refer to the *TestStand Help* for more information about the Edit Statement Step dialog box.

The Statement step type does not define any additional step properties other than the custom properties that are common to all steps.

Label

Use a Label step as the target for a Goto step. Label steps allows you to rearrange or delete other steps in a sequence without having to change the specification of targets in Goto steps.

Label steps do not pass or fail, and by default do not record results. After a Label step executes, the TestStand Engine sets the step status to `Done` or `Error`. You can edit a Label step to specify a description that appears next to the Label step name in the sequence editor.

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

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

Goto

Use Goto steps to set the next step that the TestStand Engine executes. You usually use a Label Step as the target of a Goto step. Use of a Label Step allows you to rearrange or delete other steps in a sequence without having to change the specification of targets in Goto steps.

Refer to the *TestStand Help* for more information about the Edit Goto Step dialog box.

Message Popup

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

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

Refer to the *TestStand Help* for more information about the Configure Message Popup Step dialog box.

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

- **Step.Result.ButtonHit**—Contains the one-based index of the button that you select.
- **Step.Result.Response**—Contains the response text that the you enter.
- **Step.TitleExpr**—Contains the expression for the string that appears as the title of the message popup dialog box.
- **Step.MessageExpr**—Contains the expression for the string that appears as the text message in the message popup dialog box.
- **Step.Button1Label**, **Button2Label**, **Button3Label**, **Button4Label**, **Button5Label**, and **Button6Label**—Specify the expression for the label text for each button.
- **CenterDialog**—Specifies that the message popup dialog box appears in the center of the screen.
- **Position.Top** and **Position.Left**—Specify the location of the message popup dialog box when CenterDialog is `False`.
- **Modal**—Specifies whether the message popup dialog box is a modal dialog box.
- **Step.ShowResponse**—Enables the response text box control in the message popup dialog box.
- **Step.MaxResponseLength**—Specifies the maximum number of characters that the operator can enter in the response text box control.
- **Step.DefaultResponse**—Contains the initial text string that the step displays in the response text box control.
- **Step.ButtonLocation**—Specifies whether to display the buttons on the bottom or side of the message popup dialog box.
- **Step.ActiveCtrl**—Identifies one of the six buttons or the input string as the active control.
- **Step.DefaultButton**—Specifies which button, if any, uses <Enter> as its shortcut key.
- **Step.CancelButton**—Specifies which button, if any, uses <Esc> as its shortcut key.
- **Step.TimerButton**—Specifies the index of the button that activates automatically after a timeout elapses. A value of zero indicates that no timeout occurs.
- **Step.TimeToWait**—Specifies the number of seconds before the button that Step.TimerButton specifies activates.

Call Executable

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

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

Refer to the *TestStand Help* for more information about the Configure Call Executable dialog box.

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

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

Property Loader

Refer to Appendix D, *Database Step Types*, for more information about the Property Loader step. Refer to the *TestStand Help* for more information about the Edit Property Loader dialog box.

5

Module Adapters

This chapter describes the TestStand module adapters.

Overview

The TestStand Engine uses a module adapter to invoke the code in a code module, which is called from a TestStand sequence. Each module adapter supports one or more specific types of code modules, which include LabVIEW VIs; LabWindows/CVI functions in source files, object files, or library modules that you create in LabWindows/CVI or other compilers; C/C++ functions in DLLs; .NET assemblies; ActiveX automation servers; and HTBasic subroutines. A module adapter knows how to load and call a code module, how to pass parameters to a code module, and how to return values and status from a code module.

When you edit a step that uses a module adapter, TestStand relies on the adapter to display the Specify Module dialog box, in which you specify the code module for the step and also specify any parameters to pass when you invoke the code module.

TestStand stores the name and location of the code module, the parameter list, and any additional options as properties of the step. TestStand hides most of these adapter-specific step properties.

In some cases, if the module adapter is specific to an ADE, the adapter knows how to open the ADE, how to create source code for a new code module in the ADE, and how to display the source for an existing code module in the ADE. Some adapters support stepping into the source code in the ADE while you execute the step from the TestStand Sequence Editor or Operator Interfaces.

Configuring Adapters

You can configure most of the module adapters by selecting **Configure» Adapters** from the sequence editor menu. Refer to the *TestStand Help* for more information about configuring adapters.

Source Code Templates

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

TestStand includes default templates for each of the built-in step types. You can also create additional templates for built-in step types when you create a new step type. Refer to Chapter 13, *Creating Custom Step Types*, for more information about creating code templates for step types.

LabVIEW Adapter

The LabVIEW Adapter allows you to call LabVIEW VIs with a variety of connector panes. Refer to the *Using LabVIEW with TestStand* manual for complete information about using the LabVIEW Adapter.

LabWindows/CVI Adapter

The LabWindows/CVI Adapter allows you to call C functions with a variety of parameter types. The function can exist in an object file, library file, or DLL. The function can also exist in a source file that is located in the project that you are currently using in the LabWindows/CVI development environment. Refer to the *Using LabWindows/CVI with TestStand* manual for complete information about using the LabWindows/CVI Adapter.

C/C++ DLL Adapter

The C/C++ DLL Adapter allows you to call C functions and C++ methods in a DLL with a variety of parameter types. In C++ DLLs, the methods can be either global static methods or static class methods. You can create the DLL code module with Microsoft Visual Studio .NET or any other ADE that creates a C/C++ -callable DLL.

Additionally, if you have National Instruments Measurement Studio 7.0 (or later) Enterprise Edition and Visual Studio .NET 2003 or later installed, you can create and edit C++ code modules directly from TestStand.

Note National Instruments recommends using the LabWindows/CVI Adapter to call functions in DLLs that you create using LabWindows/CVI. The LabWindows/CVI Adapter provides full integration with the LabWindows/CVI ADE for code generation and debugging.

Specifying a C/C++ DLL Adapter Module

The Specify Module dialog box for the C/C++ DLL Adapter is called the Edit C/C++ DLL Call dialog box. The Edit C/C++ DLL Call dialog box contains a Module tab and a Source Code tab.

Refer to the *TestStand Help* for more information about the Edit C/C++ DLL Call dialog box.

Debugging DLLs

To debug a DLL, first create the DLL with debugging enabled in your ADE. Then, launch the sequence editor or operator interface from LabWindows/CVI or Visual Studio, or attach to the sequence editor or operator interface process from your ADE. In LabWindows/CVI, select **Run»Select External Process** in the Project window to identify the executable for the sequence editor or operator interface. Then, use the `Run` command to start the executable.

Note If you use LabWindows/CVI or Visual Studio .NET to debug a DLL in a TestStand process, be sure to save your sequence files before you stop debugging. If you stop debugging, the ADE will terminate the TestStand process prematurely.

If you are debugging a DLL in the Test Stand process using LabWindows/CVI or Visual Studio .NET 2003 or later and you have Measurement Studio 7.0 (or later) Enterprise Edition installed, you can click **Step Into** in TestStand on a step that calls into the DLL to suspend the ADE at the first statement in the DLL function.

Table 5-1 describes your options for stepping out of a LabWindows/CVI or Visual Studio .NET DLL function that you are debugging.

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

ADE Command for Stepping Out	Result in TestStand
Finish Function or Step Out	Execution of the function. 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. If the Step Over command executes on an END step in a Pre-Step callback, TestStand attempts to Step Into the code module.
Continue	TestStand does not suspend execution when the function call returns.

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

Debugging LabVIEW DLLs You Call with the C/C++ DLL Adapter

You must use the LabVIEW Operator Interface to debug any VIs that you build into a DLL using LabVIEW 6.1 or later. First, open the operator interface in the LabVIEW development environment. Before executing the operator interface, open the VI that represents the DLL function to debug and place a break point in the diagram of this VI. Next, use the LabVIEW Operator Interface to load and execute the sequence file that calls the LabVIEW DLL. When LabVIEW loads the DLL that the step calls, LabVIEW uses the VI in memory instead of the VI in the DLL. Then, when the step calls the DLL function, LabVIEW suspends at the breakpoint that you set in the VI.

Using MFC in a DLL

The Microsoft Foundation Class (MFC) Library places several requirements on DLLs that use the DLL version of the MFC run-time library. If you call any of these DLLs, verify that the DLL meets these requirements. Also, if the DLL uses resources such as dialog boxes, verify that the `AFX_MANAGE_STATE` macro appears at the beginning of the function body of each function that you call. Refer to your MFC documentation for more information about calling DLLs.

Loading Subordinate DLLs

TestStand directly loads and runs the DLLs that you specify in the Specify Module dialog box for the C/C++ DLL Adapter. Since your code modules most likely call subsidiary DLLs, such as instrument drivers, ensure that the operating system can find and load those subsidiary DLLs.

The operating system searches for the DLLs using the following search directory precedence:

1. The directory in which the application resides.
2. The current working directory.
3. The Windows System directory.
4. The Windows directory.
5. The directories listed in the PATH environment variable.

Creating Type Libraries

A type library exposes the function names and arguments of a DLL or COM object. Typically, type libraries are part of the DLL itself. TestStand does not require you to create a DLL with a built-in type library. However, if you include a type library in your DLL, the C/C++ DLL Adapter can use the information in the type library to obtain the parameter list information.

Refer to the NI Developer Zone online at `ni.com/zone` for additional information about building type libraries.

.NET Adapter

The .NET Adapter allows you to call .NET assemblies written in any .NET-compliant language, such as C# or Microsoft Visual Basic .NET.

Note You must have the .NET Framework 1.1 or later installed in order to use the .NET Adapter.

Additionally, if you have Measurement Studio 7.0 (or later) Enterprise Edition and Visual Studio .NET 2003 or later installed, you can create and edit .NET code modules directly from TestStand.

Within the .NET assembly, you can call methods and access properties or fields on a class or struct. With an instance of a class that was either previously created and stored in an object reference variable or created

in the calling step, you can call or access all non-static public members. An instance is not required to call or access static public members.

When calling a struct, the definition can be stored in a variable of a data type that is mapped to the struct members or initialized in the calling step.

Refer to the *TestStand Help* for more information about configuring calls to .NET assemblies.

Debugging .NET Assemblies

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

Note When you debug an assembly in a TestStand process and you stop debugging, Visual Studio .NET terminates the TestStand process prematurely. Save your sequence files and workspace before you stop debugging and terminate the process.

If you are debugging a .NET assembly in the Test Stand process using Visual Studio .NET 2003 or later and you have Measurement Studio 7.0 (or later) Enterprise Edition installed, you can click **Step Into** in TestSTand on a step that calls into the assembly to suspend Visual Studio .NET at the first statement in the assembly method or property.

Table 5-2 describes your options for stepping out of a Visual Studio .NET assembly that you are debugging.

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

Visual Studio .NET Command for Stepping Out	Result in TestStand
Step Out	Executes the function. 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	Suspends execution on the next step in the sequence when you use this command on the last executable statement of the function.
Continue	Does not suspend execution when the function call returns.

Refer to your Microsoft Visual Studio .NET documentation for more information about debugging assemblies in an external process.

Note If you are using LabWindows/CVI to debug a DLL in the TestStand process, you cannot debug a .NET assembly at the same time. If you are using Visual Studio .NET 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 out-of-process.

Note When you debug a TestStand process with Visual Studio .NET, TestStand will not unload assemblies when you select **File»Unload All Modules**.

Configuring the .NET Adapter

Use the .NET Adapter Configuration dialog box to configure the .NET Adapter. Refer to the *TestStand Help* for more information about the .NET Adapter Configuration dialog box.

Numeric Parameters

The .NET Adapter supports most .NET numeric data types. TestStand does not support 96-bit floating point precision numbers. If you use the Decimal data type, it will be stored in a double and could lose precision.

Enumeration Parameters

Use the ring control in the Value column of the parameter control on the Specify Module dialog box to select one of the enumerations or enter an expression for a number or string. If the .NET Adapter enumeration uses the Flag attribute, you can pass multiple values of the bit field by separating each element in a string, such as "`enum 1, enum 2`".

Struct Parameters

You can pass variables and properties that you create with named data types to an assembly that accept structs. When you create or edit a data type, you specify whether the type can be a struct argument and how its properties map to the struct. The mapping is done between the struct element name and the name of a property of the data type. The mapping eliminates the need to specify an argument for each element of the struct. To create a new data type and map it directly to the struct, click the **Data Type** button in the Type column of the Specify Module dialog box, which launches the Create Custom Data Type From Struct dialog box.

The struct mapping of a TestStand variable will be automatically refreshed before it is used. If there is no mapping for a struct element, the argument for this element will have a question mark (`?`) in the Specify Module dialog

box. You must edit the named data type to map the struct element to a property of the type.

Array Parameters

You can specify an array that contains elements of any type. For numeric arrays, TestStand reformats the contents of the numeric array argument into a temporary array that contains elements that have the data type the assembly expects. For arrays of structs, you must specify an array of containers. The data type of the array of containers must have struct passing enabled. It must also match the struct structure expected by the arrays of structs.

For inputs, the number of dimensions in the TestStand array must match the number of dimensions in the .NET array. For outputs, the TestStand array resizes to match the .NET array.

For small arrays, you can specify an argument for each element of the array. To add elements to the array, click the <+> sign in the Type column. TestStand inserts a new element below the selected element. Click the <–> sign to delete the selected array element.

ActiveX/COM Adapter

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

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

Running and Debugging ActiveX Automation Servers

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

When you work in Microsoft Visual Basic, place breakpoints in your automation server source code and select **Run»Start with Full Compile**.

In TestStand, run the sequence that calls into your automation server to cause the execution to automatically suspend at the breakpoint that you set in Visual Basic. Refer to your ADE documentation for more information about debugging ActiveX automation servers.

Configuring the ActiveX/COM Adapter

Use the ActiveX/COM Adapter Configuration dialog box to configure the ActiveX/COM Adapter. Refer to the *TestStand Help* for more information about the ActiveX/COM Adapter Configuration dialog box.

Using ActiveX/COM Servers with TestStand

This section discusses using ActiveX/COM servers with TestStand.

Registering and Unregistering a Server

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

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

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

Compatibility Options for Visual Basic

If you are developing an automation server in an ADE that does not give you direct control over IDs, you must ensure that the ActiveX/COM Adapter can find the server identifiers or the names defined in the TestStand step. When you rebuild an ActiveX/COM server in Visual Basic, you can select one of three compatibility options. Depending on the level of compatibility and the changes you make to a project, Visual Basic compiles an appropriate new server, which can contain new identifiers.

To specify the level of compatibility, select **Project Properties** from the **Project** menu in Visual Basic. In the Project Properties dialog box, use the radio buttons in the Version Compatibility section on the Components tab to select the level of compatibility.

Visual Basic offers the following compatibility options:

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

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

Note National Instruments does not recommend the use of the No compatibility setting with your TestStand projects.

- **Project compatibility**—Causes Visual Basic to maintain the ID assignments that it uses to uniquely identify the type information for the server. Use this option when you have multiple projects under development within Visual Basic. The setting is not meant to assure compatibility with client applications that were not compiled in Visual Basic or projects that use early binding.

Note Only use the Project compatibility option after you have built the server DLL or executable.

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

Note National Instruments recommends that you configure the ActiveX/COM Adapter to use late binding when you create a server using the Project compatibility option.

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

Note Use the Binary compatibility option only after you have built the server DLL or executable for the first time.

When you use this option to rebuild a server, TestStand can use the IDs stored in the step without accessing the type information at run time. National Instruments recommends that you configure the ActiveX/COM Adapter to use early binding when you create a server with this option.

National Instruments makes the following additional recommendations regarding the use of the Visual Basic ActiveX/COM server in conjunction with development of sequences within TestStand. These approaches ensure that the ActiveX/COM Adapter can properly find and invoke the server after you recompile the server.

- Use the following approach while you develop and debug sequences:
  - Use the Project compatibility option to rebuild your server in Visual Basic.
  - Configure the ActiveX/COM Adapter to use late binding.
- Use the following approach when the interface for the server is completely developed and debugged:
  - Use the Binary compatibility option to rebuild your server in Visual Basic.
  - Select **Tools»Update Automation Identifiers** in the TestStand Sequence Editor to assign the new server identifiers to the steps.
  - After you properly assign the new server identifiers to the steps, you can enable the ActiveX/COM Adapter to use early binding.

For more information about creating and debugging Visual Basic ActiveX/COM servers, refer to your Visual Basic documentation and to the following Internet document:

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

HTBasic Adapter

The HTBasic Adapter allows you to call HTBasic subroutines without passing parameters directly to a subroutine. Instead, the subroutine exchanges data by calling get or set subroutines contained in an HTBasic CSUB. These subroutines use the TestStand API to get data from and set data in TestStand. For more information about using these subroutines, refer to the *Passing Data To and Returning Data From a Subroutine* section of this chapter.

Specifying an HTBasic Adapter Module

The Specify Module dialog box for the HTBasic Adapter is called the Edit HTBasic Subroutine Call dialog box. The dialog box contains controls to specify the subroutine file path, subroutine name, and other options. Refer to the *TestStand Help* for more information about the Edit HTBasic Subroutine Call dialog box.

Debugging an HTBasic Adapter Module

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

If you select **Debug»Step Into** in TestStand when an execution is currently suspended on a step that calls an HTBasic subroutine, HTBasic displays the HTBasic server window and pauses at the call of the subroutine. When suspended, press **<Alt-F1>** to single step through the subroutine. When you have finished debugging a particular subroutine, click **Continue** to resume execution and return control to TestStand. After you step out of the subroutine, TestStand suspends execution on the next step in the sequence.

For more information about debugging HTBasic programs, refer to your HTBasic documentation.

Passing Data To and Returning Data From a Subroutine

TestStand provides a library of CSUB routines that use the TestStand API to access TestStand variables and properties from an HTBasic subroutine. For more information about these subroutines, refer to the *TestStand Help*.

Sequence Adapter

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

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

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

You can define the parameters for a sequence on the Parameters tab in the Sequence File window, shown in Figure 5-1.

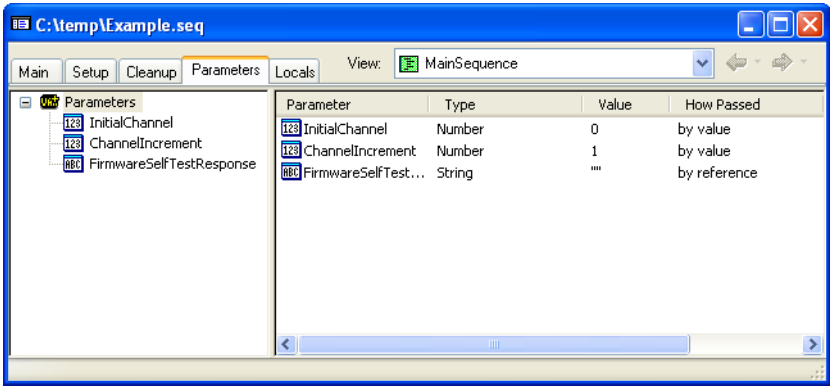


Figure 5-1. Example Sequence Parameters

The Parameters tab defines each parameter name, its TestStand data type, its default value, and whether you pass the argument by value or by reference. For more information about sequence file parameters, refer to the *TestStand Help*.

Specifying a Sequence Adapter Module

The Specify Module dialog box for the Sequence Adapter is called the Edit Sequence Call dialog box. Refer to the *TestStand Help* for more information about the Edit Sequence Call dialog box.

Remote Sequence Execution

There are three types of sequence file paths available when you use remote sequence execution—absolute, relative, and network. Table 5-3 describes these options.

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

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

When you specify a sequence file pathname in the Pathname and Sequence section of the Edit Sequence Call dialog box and specify Run Sequence on a Remote Computer in the Multithreading and Remote Execution section of the same dialog box, TestStand locates the sequence file according to the type of path, as described in Table 5-3.

When you edit a step in a sequence file on a client machine 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 machine. When you run the step on the client machine, TestStand resolves the path for the sequence file on the server system.

Choose one of the following three ways to manage your remote sequence files for remote execution:

- Add a common pathname to the search paths for the client and the server so that each resolves to the same relative pathname.
- Duplicate the files on your client and server machine so that the client edits an identical file to the file that the server runs.
- Use absolute paths that specify a mapped network drive or full network path so that the file that the client machine edits and the file the server runs are the same sequence file.

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

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

Setting up TestStand as a Server for Remote Execution

You must properly configure the server on the remote system if you want TestStand to invoke a sequence on a remote TestStand server host. These configuration settings include enabling the TestStand server to accept remote execution requests, registering the server with the operating system, and configuring Windows system security to allow users to access and launch the server remotely.

To allow the remote server to accept remote execution requests from a client machine, enable the **Allow Sequence Calls from Remote Machines to Run on This Machine** option, which is located on the Remote Execution tab on the Station Options dialog box.

A TestStand server is active while the TestStand application `<TestStand>\bin\REngine.exe` runs on a remote system. Each TestStand client communicates with a dedicated version of the remote engine application.

In Windows 2000/NT/XP, the remote server launches automatically each time a TestStand client uses the server.

In Windows 98, you must launch the remote server manually, and only one client can use the server at a time. You can automatically launch the server by placing a shortcut to the application in the `Startup` folder on the server system.

You can close remote engine applications from your system tray by enabling the **Show the System Tray Icon While the TestStand Remote System is Active on this Machine** option on the Station Options dialog box for the remote system. This option makes the TestStand icon visible in the 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.

If you do not enable this option, you must close remote engine applications manually. In Windows 2000/NT/XP, open the Windows Task Manager and select `rengine.exe` on the **Processes** tab and click **End Process**. In Windows 98, press **<Ctrl-Alt-Del>** to launch the Close Programs dialog box. Then select **REngine** from the list of active applications and click **End Task**.

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.

Before a client can communicate with a server, you must configure the security permissions for that server's Windows system.

Tip To avoid configuring security permissions, enable the **Allow All Users Access From Remote Machines** option on the Station Options dialog box. When you enable this option, TestStand configures the security permissions for you by adding the name Everyone to the list of users who have permission to launch the TestStand remote server. When you disable this option, TestStand removes the name Everyone from the list. Notice that the Allow All Users Access From Remote Machines option is not available in Windows 98.

Windows XP

For Windows XP, complete the following steps to configure the security permissions for the server:

1. Log in as a user with administrator privileges.
2. Launch the Component Services window by selecting **Start»Settings» Control Panel»Administrative Tools»Component Services** or by running `dcomcnfg` from the command line.

3. In the left pane of the Component Services window, expand the tree view to show **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 on the My Computer Properties dialog box, verify that the **Enable Distributed COM on this computer** option is enabled. Click **OK** to close the dialog box.

Note Changing the value of the **Enable Distributed COM on this computer** option requires you to reboot your computer.

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

You must give permission to the appropriate users so that they can access the remote server. You should give access to everyone, but only give launch permission to appropriate users, since users who have launch permission are able to access the server. You can set these permissions in one of the following ways:

- Specify the default security on the **Default COM Security** tab on the My Computer Properties dialog box.
- Give individual users access to the server. Use the **Security** tab on the NI TestStand Remote Engine Properties dialog box to configure the permissions for a specific server.

Windows 2000/NT

For Windows 2000/NT, complete the following steps to configure the security permissions for the server:

1. Log in as a user with administrator privileges.
2. Run `dcomcnfg` from the command line, which launches the Distributed COM Configuration Properties dialog box.
3. On the **Default Properties** tab, verify that the **Enable Distributed COM on this computer** option is enabled.

Note Changing the value of the **Enable Distributed COM on this computer** option requires you to reboot your computer.

4. On the **Applications** tab, select **NI TestStand Remote Engine** and then click **Properties**.
5. On the **Identity** tab on the NI TestStand Remote Engine Properties dialog box, verify that the **Interactive User** option is selected.

You must give permission to the appropriate users so that they can access the remote server. You should give access permissions to everyone, but only give launch permission to users who should be able to launch the server. You can set these permissions in one of the following ways:

- Specify the default security on the **Default Security** tab on the Distributed COM Configuration Properties dialog box.
- Give individual users access to the server. On the **Applications** tab, select **NI TestStand Remote Engine** and then click **Properties**. Use the **Security** tab on the TestStand Remote Engine Properties dialog box to configure the permissions for a specific server.

Windows 98

For Windows 98, complete the following steps to configure the security permissions for the server:

1. Go to the **Access Control** tab on the Network Properties dialog box in the Windows Control Panel, and enable **User-level access control**.
2. Run `dcomcnfg` from the command line, which launches the Distributed COM Configuration Properties dialog box.
3. On the **Default Properties** tab, verify that the **Enable Distributed COM on this Computer** option is enabled.

Note Changing the value of the **Enable Distributed COM on this computer** option requires you to reboot your computer.

4. On the **Default Security** tab, verify that the **Enable Remote Connection** option is enabled.

Note Changing the value of the **Enable Remote Connection** option requires you to reboot your computer.

You must give permission to the appropriate users so that they can access the remote server. You can set these permissions in one of the following ways:

- Specify the default security on the **Default Security** tab on the Distributed COM Configuration Properties dialog box.
- Give individual users access to the server. To grant individual access, select the server name, **NI TestStand Remote Engine**, on the **Applications** tab and then click **Properties**. On the **Security** tab on the NI TestStand Remote Engine Properties dialog box, add users to a list for access to the server.

In addition to clients accessing remote servers, there may be some instances in which TestStand remote servers must access the client machine. For example, if 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 machine.

You do not need to configure security permissions for the client if you are running the TestStand Sequence Editor or any of the TestStand Operator Interfaces. However, if you write your own operator interface or other client application that uses the TestStand Engine for remote execution, you must either configure security permissions for the client application as described previously for TestStand, or you must add a special registry entry for the client application that grants access to the client from remote servers. The registry entry takes the following form, where `yourclient.exe` is the name of your client application:

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

6

Database Logging and Report Generation

This chapter describes the database and report generation features of TestStand. This chapter assumes that you have a basic understanding of database concepts, SQL, and your Database Management System (DBMS) client software.

Database Concepts

This section summarizes key database concepts that are important for using databases with TestStand. It also summarizes the key Windows features TestStand uses to communicate with a DBMS.

Databases and Tables

A database is an organized collection of data. You can store data in and retrieve data from a database. Although databases vary in how they store their internal data, most modern DBMSs, also known as database servers, store data in table form.

Tables contain *records*, also known as *rows*. Each row consists of *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. The available data types vary depending on the DBMS.

You can use database tables to store many different types of data. Table 6-1 contains columns for the UUT number, a step name, a step result, and a measurement. 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.

Table 6-1. Example Database Table

UUT_NUM	STEP_NAME	RESULT	MEAS
20860B456	TEST1	PASS	0.5
20860B456	TEST2	PASS	(NULL)
20860B123	TEST1	FAIL	0.1
20860B789	TEST1	PASS	0.3
20860B789	TEST2	PASS	(NULL)

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

Use an SQL SELECT command, or *query*, to retrieve records from a database. The result of a query is called a *record set* or *SQL statement data*. The data you receive does not necessarily reflect the entire contents of any particular table in the database. For instance, you can retrieve only selected columns and rows from one table, or you can retrieve data that is a combination of the contents of multiple tables. The query defines the contents and order of the data. You can refer to each column you retrieve by the name of the column or by a one-based number that refers to the order of the column in the query.

Database Sessions

Database operations occur within a database session. A simple session follows this order:

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

Microsoft ADO, OLE DB, and ODBC Database Technologies

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

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

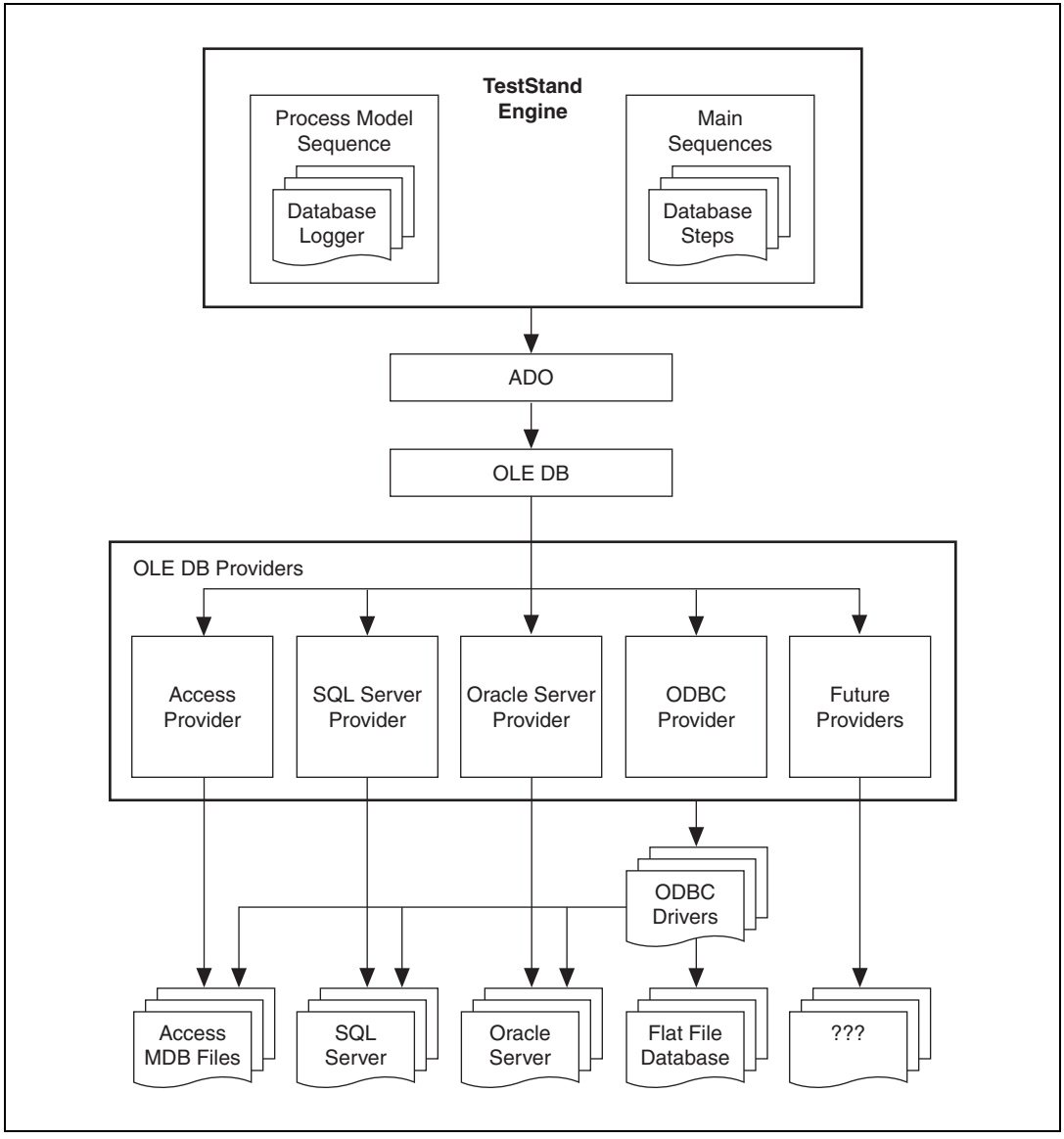


Figure 6-1. Microsoft Windows Database Technologies

Refer to the Microsoft Web site, `www.microsoft.com/data`, for more information about database technologies for Windows operating systems.

Data Links

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

For example, to connect to a Microsoft SQL Server database, specify the OLE DB provider for an SQL Server, a server name, a database name, a user ID, and a password. To connect to a Microsoft Access database, specify the OLE DB provider for ODBC and an ODBC data source name. The ODBC data source name specifies which ODBC driver to use, the database file (`.mdb`), and an optional user ID and password. You can define ODBC data source names in the ODBC Administrator in the Windows Control Panel.

Refer to the *Using the ODBC Administrator* section of this chapter for more information about the ODBC Administrator.

A *connection string* is a string version of the connection information required to open a session to a database. TestStand allows you to build a connection string using the Data Link dialog box.

The Data Link dialog box and the information contained in the connection string vary according to the OLE DB provider. For example, a connection string for a Microsoft SQL Server database might contain the following:

```
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.

Refer to the *Using Data Links* section of this chapter for more information about specifying data links. You can also refer to the *TestStand Help* for more information about the Data Link Properties dialog box.

Database Logging Implementation

TestStand's database logging capability is not native to the TestStand Engine or the TestStand Sequence Editor. The default process model included with TestStand contains customizable sequences that implement the logging features. Refer to Appendix A, *Process Model Architecture*, for more information about customizing the default process model.

The default process model relies on the automatic result collection capability of the TestStand Engine to accumulate the raw data that is logged to a database for each UUT. The engine can automatically collect the results of each step into a result list for an entire sequence, which contains the result of each step it runs and the result list of each subsequence call it makes. The default process model calls the Main sequence in the client sequence file to test a UUT, so that the result list for the Main sequence contains the raw data to log to a database for the UUT. Refer to the *Result Collection* section of Chapter 3, *Executions*, for more information about automatic result collection.

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

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

- The data link to which TestStand logs results.
- The database schema that TestStand uses. A schema contains the SQL statements, table definitions, and TestStand expressions that instruct TestStand on how to log results to a database. TestStand includes at least one set of predefined schemas and at least one schema for each supported DBMS. You can also create new schemas that log results to tables you define.
- Various filtering options to limit the amount of data that TestStand logs.
- Whether the process models log data after each step is executed or after each pass through the UUT loop.

For more information about the Database Options dialog box, refer to the *TestStand Help*.

You can also customize or replace any portion of the database logging sequences. Refer to Appendix A, *Process Model Architecture*, for more information about customizing the default process model.

Using Database Logging

Complete the following steps before using the default process model to log results to a database:

1. Decide which DBMS you want TestStand to log the results to. By default, TestStand supports Microsoft SQL Server, Oracle, and Microsoft Access. If you decide to use another DBMS, refer to the *Adding Support for Other Database Management Systems* section of this chapter.
2. Make sure that you have installed the appropriate client DBMS software that is required to communicate with the DBMS.

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

   Refer to the `<TestStand>\Doc\readme.txt` for more information about suggested providers, versions of ODBC drivers, and client DBMS software.
3. Create the default database tables in a database in your DBMS.

   TestStand comes with SQL script files for creating and deleting the default database tables that the default 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. These script files are located in the `<TestStand>\Components\NI\Models\TestStandModels\Database` directory.

   TestStand installs an example Microsoft Access database, `TestStand Results.mdb`, in the `<TestStand>\Components\NI\Models\TestStandModels\Database` directory.

   For more information about creating the default database tables using an SQL script file, refer to the *Using Data Links* section of this chapter. Refer to the *Creating the Default Result Tables* section of this chapter for more information about the default table schema used by the process model.
4. Use the Database Options dialog box to enable database logging and to define a data link and schema for the default process model to use.

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

Logging Property in the Sequence Context

When the database logger starts, it creates a temporary property named Logging in the sequence context in which the database logger evaluates expressions. The Logging property contains subproperties that provide information about database settings, process model data structures, and the results that the logger processes. As the Logging property processes the result list, the logger updates subproperties of Logging to refer to the UUT result, step result, and the step result subproperty the logger is processing. You can reference the Logging subproperties in the precondition and value expressions that you specify for statement and column values.

The Logging property has the following subproperties:

- **UUTResult**—Contains the UUT result that the logger is processing. If the logger is processing a step or a subproperty, this property holds the UUT result that contains the step result or subproperty.
- **StepResult**—Contains the step result that the logger is processing. If the logger is processing a subproperty, this property holds the step result that contains the subproperty. If the logger 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.
- **StepResultProperty**—Contains the subproperty of the step result that the logger is processing. If the logger is not processing a subproperty, this property does not exist.
- **ExecutionOrder**—Contains a numeric value that the logger increments after it processes each step result.
- **StartDate**—Specifies the date on which the UUT test began. This property is an instance of the DateDetails custom data type.
- **StartTime**—Specifies the time at which the UUT test began. This property is an instance of the TimeDetails custom data type.
- **UUT**—Specifies the serial number, test socket index, and other information about the UUT. This property is an instance of the UUT custom data type.

- **DatabaseOptions**—Contains the process model database settings you configure in the Database Options dialog box. This property is an instance of the DatabaseOptions custom data type.
- **StationInfo**—Specifies the station ID and the user name. This property is an instance of the StationInfo custom data type.

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

TestStand Database Result Tables

This section describes the default table schemas that TestStand uses. This section also outlines how to modify existing schemas or create new schemas.

Default TestStand Table Schema

The default TestStand database schema requires the following tables in your database:

- UUT_RESULT
- STEP_RESULT
- STEP_SEQCALL
- STEP_PASSFAIL
- STEP_CALLEXE
- STEP_MSGPOPUP
- STEP_PROPERTYLOADER
- STEP_STRINGVALUE
- MEAS_NUMERICLIMIT
- MEAS_IVI_WAVE
- MEAS_IVI_WAVEPAIR
- MEAS_IVI_SINGLEPOINT

The UUT_RESULT table contains information about each UUT that TestStand tests. The STEP_RESULT table contains information about each step that TestStand executes while testing each UUT. The other table names with the STEP prefix contain information for each specific step type. The table names with the MEAS prefix contain information about sub results that TestStand logs for a step type.

Each table contains a primary key column ID. The data type of the column is Number, String, or GUID, depending on the selected schema. Each table might contain foreign key columns. The data types of the columns must match the primary key that the data types reference.

Refer to the *TestStand Help* for complete information about the default TestStand table schemas.

Creating the Default Result Tables

The TestStand logging feature requires that you create the database tables in a database for your DBMS. You can use TestStand's Database Viewer to create the default result tables that the schema requires.

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

For more information about creating the default database tables using an SQL script file, refer to the *Database Viewer—Creating Result Tables* section of this chapter. Refer to the *TestStand Help* and the *Using Data Links* section of this chapter for more information about configuring your system to access your DBMS.

Adding Support for Other Database Management Systems

TestStand supports Microsoft SQL Server, Oracle, and Microsoft Access. You can add support for another DBMS in the following ways:

- Review the default schemas in the Database Options dialog box. TestStand includes schemas for each DBMS, and each schema conforms to the default database tables.
- Create result tables for the default table schema for each DBMS by using the SQL script files located in the `<TestStand>\Components\NI\Models\TestStandModels\Database` directory.

If you want to add support for another DBMS, you must create a new schema in the Database Options dialog box. Use the Duplicate button on the Schemas tab to copy an existing schema, and then customize its statement, column, and parameter settings to work with the new DBMS.

You can also follow these steps to create new script files for your new DBMS:

1. Create new script files in the `<TestStand>\Components\User\Models\TestStandModels\Database` directory.

Tip National Instruments recommends including the name of the DBMS in the filename.

2. Enter the SQL commands for creating and deleting your DBMS tables to the new script files. Refer to any of the SQL database script files that TestStand provides for an example.

   For example, the SQL database syntax file for Oracle result tables might contain the following commands:

```
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 TestStand 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 result tables:

```
DROP TABLE STEP_RESULT
/
DROP SEQUENCE SEQ_STEP_RESULT
/
DROP FUNCTION STEP_RESULT_NEXT_ID
/
```

Database Viewer

TestStand includes the Database Viewer application for viewing data in a database, editing table information, and executing SQL commands. The Database Viewer application, `DatabaseView.exe`, is located in the `<TestStand>\Components\NI\Tools\DatabaseView` directory.

For more information about the Database Viewer, refer to the *TestStand Help*.

On-The-Fly Database Logging

When you enable the **On-The-Fly Database Logging** option, the process models progressively log result data concurrent with the execution instead of waiting until the execution or testing of the UUT is complete. Database logging uses the ProcessModelPostResultListEntry and SequenceFilePostResultListEntry callbacks to process the step results. The final data logged is essentially identical to the data generated by the process model 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 is executing. Use the **Discard Results** and **Disable Results When Not Required by Model** options on the Model Options dialog box to instruct TestStand to discard step results after logging each result.

If you use On-The-Fly Database Logging with a schema that uses either stored procedure or command statements that do not use the INSERT command, you cannot define constraints for foreign keys in step result statements that references primary keys in UUT results. Defining constraints for these types of foreign keys will generate an error, since the on-the-fly database logger cannot execute the statement to create the record containing the primary key before executing the statement to create the record containing the foreign key.

Using Data Links

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

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

Using the ODBC Administrator

To access databases through the ODBC standard, you must have an ODBC driver for each database system you use. Each ODBC driver must register itself with the operating system when you install it. You must also define and name data sources in the ODBC Administrator in the Windows Control Panel. This 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.

- To access the ODBC Administrator in Windows 98, select **Start»Settings» Control Panel»ODBC Administrator**.
- To access the OBDC Administrator in Windows 2000 or Windows XP, select **Start»Settings»Control Panel»Administrative Tools»Data Sources (ODBC)**.

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

For more information about the ODBC Data Source Administrator dialog box, refer to the *TestStand Help*.

Example Data Link and Result Table Setup for Microsoft Access

This section outlines an example of how to link a TestStand data link to a Microsoft Access database file (`.mdb`) using the Microsoft Jet OLE DB provider to log results using the default process model.

Database Options—Specifying a Data Link and Schema

To configure the database logging options complete the following steps:

1. Launch the sequence editor and log in as `Administrator`.
2. Select **Configure»Database Options** to launch the Database Options dialog box. The Logging Options tab is active.
3. Enable database logging by clicking the checkbox next to the **Disable Database Logging** option.
4. Click the **Data Link** tab on the Database Options dialog box and select **Access** from the Database Management System ring control.
5. Click **Build** to launch the Data Link Properties dialog box.
6. Select **Microsoft Jet 4.0 OLE DB Provider** on the **Provider** tab on the Data Link Properties dialog box.
7. Click **Next**. The **Connection** tab is now active.
8. On the **Connection** tab, click **Browse** to launch the Select Access Database dialog box.
9. Using the Select Access Database dialog box, locate your Microsoft Access database file (`.mdb`) and click **Open** to select the file.
10. On the Data Link Properties dialog box, click **Test Connection** 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 control on 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 your database:

1. If you are continuing from the steps in the previous section, skip to Step 2. Otherwise, complete the following steps:
   a. Launch the sequence editor and log in as `Administrator`.
   b. Select **Configure»Database Options** to launch the Database Options dialog box. The Logging Options tab is active.
   c. Click the **Data Link** tab on the Database Options dialog box.

2. Select **View Data** to open the data link in the Database Viewer application.

Note Step 2 requires that the Connection String control contains a valid expression.

3. Select **File»New Execute SQL Window** to open an Execute SQL window.
4. Select **SQL»Load From File** and select the `Access Create Generic Recordset Result Tables.sql` file in the `<TestStand>\Components\NI\Models\TestStandModels\Database` directory.

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

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

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

The remainder of this chapter describes how to manage and use test reports in TestStand.

Implementation of the Test Report Capability

Most of the test report capabilities described in this chapter are not native to the TestStand Engine or the TestStand Sequence Editor. Instead, the default process model that comes with TestStand implements the test report capabilities. This approach allows you to customize all aspects of test reporting. Refer to Appendix A, *Process Model Architecture*, for more information about the default process model.

Even if you do not modify or replace the test report implementation in the process model, you can still customize the contents of test reports using the Report Options dialog box provided in the default process model. Refer to the *TestStand Help* for more information about the Report Options dialog box.

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

Using Test Reports

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

In the TestStand Sequence Editor, the Report tab on the Execution window displays the report for the current execution. Usually, the Report tab is empty until execution completes. The default process model can generate reports in either XML, HTML, or ASCII-text formats.

You can also use an external application to view reports in these or other formats by selecting **View»Launch Report Viewer** when an Execution window is active. Select **Configure»External Viewers** to specify the external application that TestStand launches to display a particular report format.

Refer to the *TestStand Help* for more information about the Report Options and Configure External Viewers dialog boxes.

Figure 6-2 shows a test report in XML or HTML text format as it appears on the Report tab in an Execution window.

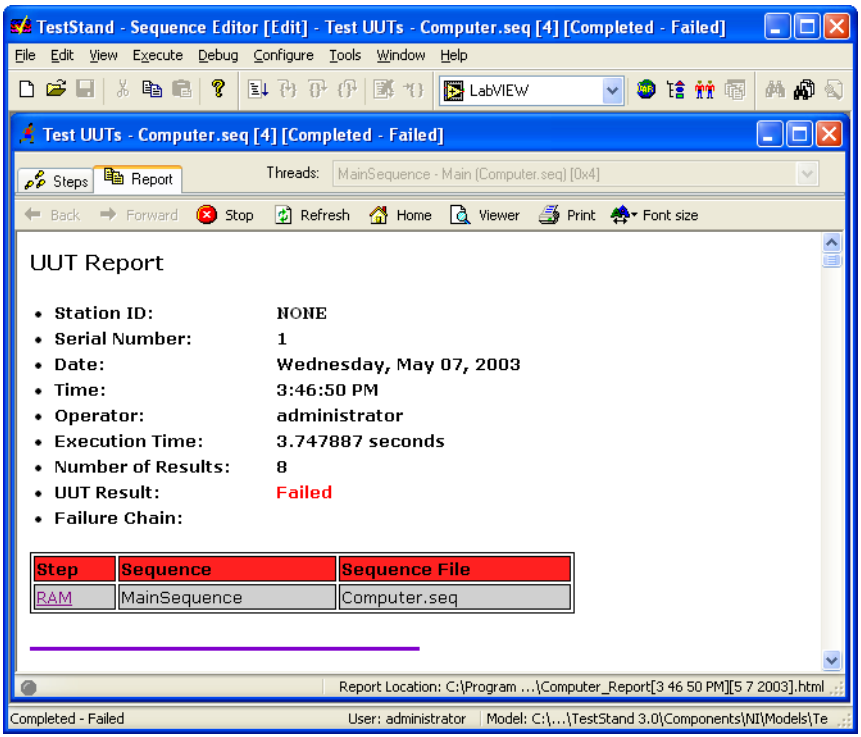


Figure 6-2. XML or HTML Test Report on the Report Tab

Figure 6-3 shows a test report in ASCII-text format as it appears on the Report tab in an Execution window.

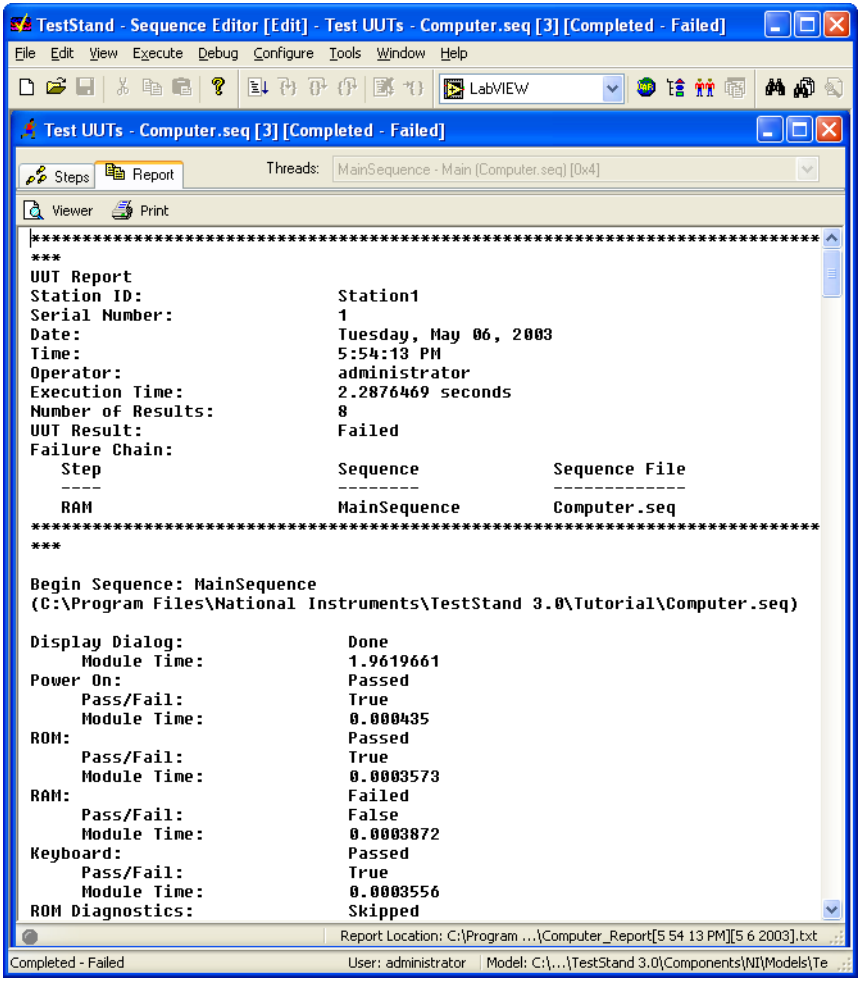


Figure 6-3. ASCII-Text Test Report on the Report Tab

Failure Chain in Reports

For UUTs that fail, XML, HTML, and ASCII-text reports include a failure chain section in the report header. The first item in the failure chain table shows the step whose failure causes the UUT to fail. The remaining items show the Sequence Call steps through which the execution reaches the failing step. In XML and HTML reports, each step name in the failure chain is a hyperlink to the section of the report that displays the result for the step. Figures 6-2 and 6-3 show a failure chain in which the failure of the RAM Test step in `MainSequence` causes the UUT to fail.

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. When the report format is XML or HTML, the batch report provides hyperlinks to each UUT report. Figure 6-4 shows an example batch report.

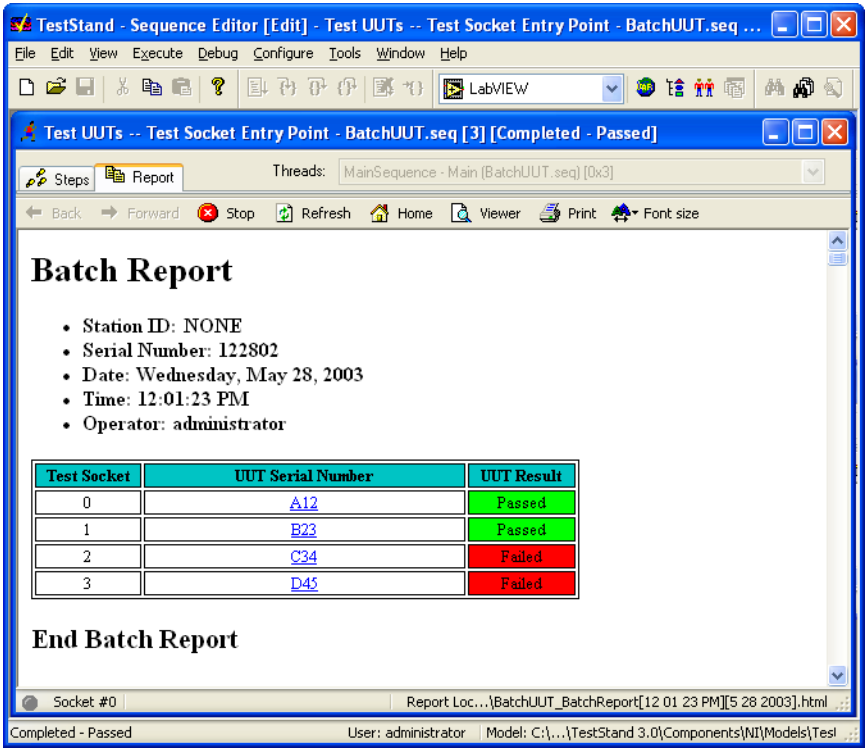


Figure 6-4. Example Batch Report

Property Flags that Affect Reports

TestStand includes three flags that you can set to identify the result properties that should be automatically displayed in the report:

- PropFlags_IncludeInReport
- PropFlags_IsLimit
- PropFlags_IsMeasurementValue

The IncludeInReport flag specifies that a subset of the result properties should automatically appear in the report. For properties that hold output values or limit values, use the IsLimit and IsMeasurementValue flags to selectively exclude limits or output values according to the option you select in the Report Options dialog box. If an array or container property sets a reporting flag, the report generator also considers the flag to be set for all array elements or subproperties within the containing object.

Tip When you use the IncludeInReport, IsLimit, and IsMeasurementValue flags on result properties for your custom step types, the report generator will format those properties into the report. If you require specific formatting for your report, use these flags to achieve the report output you want without altering the report generator settings.

On-The-Fly Report Generation

When you enable the On-The-Fly Reporting option, which is located on the Contents tab on the Report Options dialog box, the process models will progressively generate the test report concurrent with the execution instead of waiting until the execution or the testing of UUTs is complete. The final test report generated by the On-The-Fly Report Generator is identical to that generated by the process models at the end of execution.

When you use on-the-fly reporting, you can select the Report tab in the Execution window to view the test report during the execution. If the Report tab is the active view of the Execution window while a sequence is executing, the test report will periodically update as step results are processed.

In addition to generating the test report concurrently with execution, the On-The-Fly Report Generator periodically persists the current test report to a temporary file. The persistence interval is specified in the process model sequences. The temporary file is deleted and the final test report is saved to file at the end of a UUT loop execution. For more information about process model sequences, refer to Appendix A, *Process Model Architecture*.

If the Conserve Memory and Only Display Latest Results report option is enabled, the On-The-Fly Report Generator periodically purges internal data-structures. As a result, the test report that is displayed in the Report view of the Execution window will only show the results for those steps that have not yet been purged. The persisted temporary and final test report files, however, will contain all of the step results. For these files, the step results for Sequence Call and Loop Result steps will appear after their corresponding Sequence Call and Loop Index step results, respectively.

Use the Discard Results or Disable Results When Not Required By Model options on the Model Options dialog box to instruct TestStand to conserve memory by discarding results after reporting each result.

XML Report Schema

XML Test Reports are described according to the XML W3C Schema. The XML Schema Definition (XSD) file is stored in the following location: `<TestStand>\Components\NI\Models\TestStandModels\Report.xsd`.

7

User Management

The TestStand Engine features a *user manager*, which is a list of users, their user names and passwords, and their privileges. TestStand limits the functionality of the sequence editor and operator interfaces depending on the privilege settings that have been defined in the user manager for the current user.

When you launch the TestStand Sequence Editor or any of the TestStand Operator Interfaces, they display the Login dialog box by calling the LoginLogout Front-End callback sequence. The LoginLogout sequence calls the DisplayLoginDialog method of the Engine class, which launches the actual Login dialog box.

The User Manager tab on the Station Options dialog box specifies whether TestStand enforces user privileges and specifies the location of the user manager configuration file. Refer to the *TestStand Help* for more information about the User Manager tab on the Station Options dialog box.

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

For information about the User Manager window, adding users, and setting user privileges, refer to the *TestStand Help*.

Verifying User Privileges

This section discusses how to verify privileges by user.

Accessing Privilege Settings for the Current User

To verify in an expression that the current user has a specific privilege, call the `CurrentUserHasPrivilege` expression function. To verify the privilege in a code module, call the `CurrentUserHasPrivilege` method of the `Engine` class in the TestStand API.

When you call the CurrentUserHasPrivilege expression function or method, you must specify the property name of the privilege as a string argument. The current user has a privilege if the property is `True` or if the GrantAll property in any enclosing privilege group is `True`. For example, a user has the privilege to terminate an execution if the User.Privileges.Configure.Terminate property is `True`, if the User.Privileges.Configure.GrantAll property is `True`, or if the User.Privileges.GrantAll property is `True`. The CurrentUserHasPrivilege expression function returns `True` if the current user has the privilege or you have disabled privilege checking.

You can pass any subset of the property name tree structure to the CurrentUserHasPrivilege expression function. For example, you can use either of the following two expressions to determine whether the current user has the privilege to terminate an execution:

- CurrentUserHasPrivilege("Terminate")
- CurrentUserHasPrivilege("Configure.Terminate")

You can pass `"*"` as the string argument to the CurrentUserHasPrivilege expression function to determine whether a user is currently logged in. Refer to Chapter 1, *TestStand Architecture*, for more information about using expressions.

The CurrentUserHasPrivilege method behaves identically to the expression function, except that it takes additional parameters. For more information about Engine.CurrentUserHasPrivilege, refer to the *TestStand Help*.

Accessing Privilege Settings for Any User

The TestStand API includes methods that give you access to the privileges of any user. Use `Engine.GetUser` to return a User object. You can then use `User.HasPrivilege` to inspect the value of a specific privilege. This method behaves identically to the CurrentUserHasPrivilege expression function. Refer to the *TestStand Help* for more information about the User.HasPrivilege method.

8

Customizing and Configuring TestStand

This chapter describes how to configure and customize a TestStand station.

Customizing TestStand

This section describes the various TestStand components that you can customize to meet your specific needs.

Operator Interfaces

The TestStand Operator Interfaces are application programs that you use to execute and debug test sequences on a test station. The operator interfaces are available in several different programming languages and include full source code, allowing you to modify them to meet your specific needs.

Refer to Chapter 9, *Creating Custom Operator Interfaces*, for more information about how to create and modify TestStand Operator Interfaces.

Process Models

The TestStand process models define the set of operations that occur for all test sequences, such as identifying the UUT, notifying the operator of pass/fail status, generating a test report, and logging results. TestStand includes three fully customizable process models to meet your specific testing needs: Sequential, Parallel, and Batch.

Refer to Chapter 10, *Customizing Process Models and Callbacks*, to learn how to modify the TestStand process models.

Callbacks

TestStand calls callback sequences at specific points during sequence execution and test station operation. You can modify these callbacks to customize the operation of your test station.

Refer to Chapter 10, *Customizing Process Models and Callbacks*, to learn how to modify TestStand callback sequences.

Data Types

Data types define station global variables, sequence file global variables, sequence local variables, and properties of steps and step types. You can create and modify your own data types in TestStand, as well as modify the standard TestStand data types.

Refer to Chapter 11, *Type Concepts*, and Chapter 12, *Standard and Custom Data Types*, to learn how to create and modify TestStand data types.

Step Types

Steps that you add to TestStand sequences are instances of step types. A step type defines the behavior and properties of a step. You can create and modify your own step types in TestStand, as well as modify the standard TestStand step types.

Refer to Chapter 11, *Type Concepts*, and Chapter 13, *Creating Custom Step Types*, to learn how to create and modify TestStand step types.

Tools Menu

The TestStand Sequence Editor and Operator Interfaces each include a Tools menu that contains common tools for use with TestStand. These tools include a documentation generator, converters for LabVIEW and LabWindows/CVI Test Executive sequences, a Database Viewer application, and the TestStand Deployment Utility. You can modify the Tools menu to contain the exact tools you need. You can also add new items to the Tools menu.

Refer to the *TestStand Help* for more information about how to add your own commands to the Tools menu using the Customize Tools Menu dialog box.

TestStand Directory Structure

The TestStand installation program installs the TestStand Engine, the TestStand Sequence Editor, the module adapters, and additional components on your system. Table 8-1 shows the names and contents of each subdirectory of the TestStand installation.

Table 8-1. TestStand Subdirectories

Directory Name	Contents
`AdapterSupport`	Support files for the LabVIEW, LabWindows/CVI, and HTBasic Adapters.
`API`	TestStand ActiveX automation server libraries and utility libraries for several programming languages.
`Bin`	TestStand Sequence Editor executable, TestStand Engine DLLs, and support files.
`Cfg`	Configuration files for TestStand Engine and TestStand Sequence Editor options.
`CodeTemplates`	Source code templates for step types.
`Components`	Components that are installed with TestStand and components that you develop. This includes callback files, converters, icons, language files, process model files, step types, source files, and utility files.
`Doc`	Documentation files.
`Examples`	Example sequences and tests.
`OperatorInterfaces`	LabVIEW, LabWindows/CVI, Microsoft Visual Basic, C#, and C++ (MFC) Operator Interfaces with source code.
`Redist`	Redistributable engine component for use with the TestStand Deployment Utility.
`Setup`	Support files for the TestStand installer.

NI and User Subdirectories

Three of the TestStand directories contain source files that you can modify or replace: `OperatorInterfaces`, `CodeTemplates`, and `Components`. These directories contain `NI` and `User` subdirectories.

By default, TestStand installs its files into the `NI` subdirectory. Use the `User` subdirectory to store your modified files to ensure that installations of newer versions of TestStand do not overwrite your customizations. The `User` subdirectory also acts as the staging area for the components that you include in your own run-time deployment of TestStand.

The Components Directory

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

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

The TestStand Engine searches for sequences and code modules using the TestStand search directory path. The default search precedence places the `<TestStand>\Components\User` directory tree before the `<TestStand>\Components\NI` directory tree. This ensures that TestStand loads the sequences and code modules that you customize instead of loading the default TestStand versions of the files. To modify the precedence of the TestStand search directory paths, select **Configure» Search Directories** from the sequence editor menu bar.

When you deploy a run-time version of the TestStand Engine, you can bundle your components in the `User` directory with the TestStand run-time deployment. Refer to Chapter 14, *Deploying TestStand Systems*, for more information about how to deploy the TestStand Engine and your custom components.

Table 8-2 lists each subdirectory found in the NI and User directory trees of the <TestStand>\Components directory.

Table 8-2. TestStand Component Subdirectories

Directory Name	Contents
Callbacks	The Callbacks directory contains the sequence files in which TestStand stores Station Engine callbacks and Front-End callbacks. TestStand installs the Station Engine and Front-End callback files into the <TestStand>\Components\NI\Callbacks directory tree. Refer to Chapter 10, *Customizing Process Models and Callbacks*, for more information about customizing the Station Engine and Front-End callbacks.
Icons	The Icons directory contains icon files for module adapters and step types. TestStand installs the icon files for module adapters and built-in step types into the <TestStand>\Components\NI\Icons directory. Refer to Chapter 13, *Creating Custom Step Types*, for more information about creating your own icons for your custom step types.
Language	The Language directory contains string resource files. It has one subdirectory per language. Refer to the *Creating String Resource Files* section of this chapter for more information about creating string resource files in the Language directory tree.
Models	The Models directory contains the default process model sequence files and supporting code modules. Refer to Chapter 10, *Customizing Process Models and Callbacks*, for more information about customizing the process model.
RuntimeServers	The RuntimeServers directory contains a LabVIEW 6.1 run-time application for executing LabVIEW 6.1-based code modules without the LabVIEW development system. Refer to Chapter 5, *Configuring the LabVIEW Adapter*, of *Using LabVIEW with TestStand* for more information about using LabVIEW run-time servers.
StepTypes	The StepTypes directory contains support files for step types. TestStand installs the support files for the built-in step types into the <TestStand>\Components\NI\StepTypes directory tree. Refer to Chapter 13, *Creating Custom Step Types*, for more information about customizing your own step types.
Tools	The Tools directory contains sequences and supporting files for the Tools menu commands. Refer to the *Tools Menu* section of this chapter for more information about customizing the Tools menu.

Creating String Resource Files

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

The directory search follows this order:

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

To change the current language setting, select **Configure»Station Options**.

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

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

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

Resource String File Format

Each string resource file must have the `.ini` file extension. String resource files use the following format:

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

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

Note When you create new entries in a string resource file, begin your category name with a unique ID such as a company prefix. Using a unique ID will prevent name collision. For example, `NI_SUBSTEPS` uses `NI` as a unique ID.

When you specify custom resource strings, you create the category and tag names. The number of categories and tags is unlimited. A string can be of unlimited size. However, if a string has more than 512 characters, you must break it into multiple lines. Each line has a tag suffix of `line`*NNNN*, where *NNNN* is the line number with zero padding. The following is an example of a multiple-line string:

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

You can use escape codes to insert unprintable characters. Table 8-3 lists the available escape codes.

Table 8-3. Resource String File Escape Codes

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

The following string shows how to use `\n`, the embedded linefeed character:

```
tag1 = "This is line one.\nThis is line two"
```

Configuring TestStand

This section outlines the configuration options in TestStand.

Sequence Editor and Operator Interface Startup Options

You can append certain options to the sequence editor and operator interface command line, separating various parameters by spaces. The sequence editor and all operator interface applications support command-line options for opening and running sequences. Table 8-4 shows the valid startup options.

Table 8-4. Sequence Editor or Operator Interface Startup Options

Option	Purpose
sequencefile {sequencefile2}...	Instructs the application to automatically load the sequence files at startup. For example: `SeqEdit.exe "c:\MySeqs\seq1.seq" "c:\MySeqs\seq2.seq"`
`/run` *sequence sequencefile*	Instructs the application to automatically load and run the sequence in the sequence file at startup. For example: `SeqEdit.exe /run MainSequence "c:\MySeqs\test.seq"`
`/runEntryPoint` *entrypointname sequence file*	Instructs the application to automatically load and execute the sequence file at startup using the specified Execution entry point. For example: `SeqEdit.exe /runEntryPoint "Test UUTs" "c:\MySeqs\test.seq"`

Table 8-4. Sequence Editor or Operator Interface Startup Options (Continued)

Option	Purpose
`/quit`	Instructs the application to exit after the specified automatically-run executions complete. For example: `SeqEdit.exe /run MainSequence "c:\MySeqs\test\seq" /quit` Notice that the `/quit` command is ignored if the execution fails to launch.
`/useExisting`	Instructs the application to not open a new instance of the sequence editor if one is already running. For example: `SeqEdit.exe /useExisting` Notice that this option is only available for use with the sequence editor. Additionally, the sequence editor will ignore this option if the `/quit` option is specified.
`/setCurrentDir`	Instructs the application to set the current directory to the first directory in the file dialog directory history list. The current directory is the directory that the File dialog box initially displays when you open or save a file. This option allows you to set the directory displayed by the File dialog box to the directory that was displayed in the File dialog box the last time that you ran the sequence editor. The sequence editor sets the current directory after processing the other command line options. For example: `SeqEdit.exe /setCurrentDir`
`/?`	Instructs the application to launch a help dialog box that contains a list of valid command-line arguments, and then to close immediately. For example: `SeqEdit.exe /?` Notice that all other options are ignored if the `"/?"` option is specified.

Note Both `"/"` and `"-"` are valid command prefixes.

Note Quotes are required for arguments that contain spaces, such as `"Test UUTs"` and `"C:\My Documents\MySeq.seq"`.

Note If there is an error in the command-line argument, the Application Manager control generates a ReportError() event, which allows you to resolve the invalid command lines.

Configure Menu

The Configure menu in the sequence editor and operator interfaces contains commands that control the operation of the TestStand station. The following section provides a brief overview of the items in the Configure menu. Refer to the *TestStand Help* for more information about the dialog boxes that each menu item launches.

- **Sequence Editor Options**—Launches the Sequence Editor Options dialog box, in which you can set preferences for the sequence editor.
- **Station Options**—Launches the Station Options dialog box, in which you can set preferences for your TestStand station. These settings affect all sequence editor and operator interface sessions that you run on your computer.
- **Search Directories**—Launches the Search Directories dialog box, in which you can customize the search paths for finding files. The Search Directories dialog box also displays a list of paths. The paths that appear first in the list take precedence over the paths that appear later. When you first run TestStand, the list contains a default set of directory paths.
- **External Viewers**—Launches the Configure External Viewers dialog box, in which you can specify the external viewer to use for each report format.
- **Adapters**—Launches the Adapter Configuration dialog box, in which you can configure a specific module adapter, specify the active module adapter, or configure whether the adapter is hidden in the Adapter ring control in the sequence editor.
- **Report Options**—Launches the Report Options dialog box, in which you can customize the generation of report files.
- **Database Options**—Launches the Database Options dialog box, in which you can customize the logging of test result data.
- **Model Options**—Launches the Model Options dialog box, in which you can specify process model specific options such as the number of test sockets in the system.

9

Creating Custom Operator Interfaces

This chapter outlines how to create or customize an operator interface application. It also describes the various example operator interface applications that TestStand provides.

Refer to the following documents and examples in preparation for creating a custom operator interface application:

- The *Writing an Application with the TestStand UI Controls* section of this chapter.
- The following sections of the *TestStand Help:*
  - *TestStand ActiveX API Overview*—Contains an overview of the TestStand ActiveX Server functionality and discusses how to call the API from different programming languages.
  - *Core UI Classes, Properties, Methods, and Events*—Lists the core classes in the TestStand UI Controls.
  - *Core API Classes, Properties, and Methods*—Lists the core classes in the TestStand API.
- The *TestStand User Interface Controls Reference Poster*, which is included in your TestStand package.
- The information in Chapter 6, *Creating Custom User Interfaces in LabVIEW*, of the *Using LabVIEW with TestStand* manual and in Chapter 6, *Creating Custom User Interfaces in LabWindows/CVI*, of the *Using LabWindows/CVI with TestStand* manual.

  If you are using an environment other than LabVIEW or LabWindows/CVI, you can still refer to one of these sources for general instructions on how to construct an operator interface application.
- The example projects and source code located in the `<TestStand>\OperatorInterfaces\NI` directory. Start with these examples and customize them to meet your requirements.

Note The TestStand UI Controls are not supported in Windows 98.

Example Operator Interfaces

TestStand installs the executable, project, and source files for each example operator interface in the <TestStand>\OperatorInterfaces\NI directory. This directory contains the Full-Featured and Simple subdirectories. The Full-Featured subdirectory contains operator interfaces that allow you to load, view, execute, and debug sequence files. The Simple subdirectory contains similar operator interfaces that are more limited than those in the Full-Featured subdirectory in that they have no menus and fewer commands and options. Also, the simple examples do not display steps for sequences you load, but they do display the steps for executions you run.

Both subdirectories contain examples with source code for LabVIEW, LabWindows/CVI, C#, Microsoft Visual Basic .NET, and C++ (MFC).

To customize one of these operator interfaces, copy the operator interface project and source files from the NI subdirectory to the <TestStand>\OperatorInterfaces\User subdirectory before customizing them to ensure that a newer installation of TestStand does not overwrite your customizations.

Tip National Instruments recommends that you track the changes you make to the operator interface source so that you can integrate your changes with any enhancements from future versions of the TestStand Operator Interfaces.

TestStand User Interface Controls

With the exception of the examples located in the <TestStand>\OperatorInterfaces\NI\TestStand 2.0.1 Operator Interfaces (Old) directory, all operator interface examples use the TestStand User Interface (UI) Controls. The TestStand UI Controls are a set of ActiveX controls that implement the common functionality that applications need in order to display, execute, and debug test sequences. These ActiveX controls greatly reduce the amount of source code an operator interface application requires.

National Instruments recommends that you use these controls to develop your operator interface applications. However, you can also create an application by directly calling the TestStand API. The examples in <TestStand>\OperatorInterfaces\NI\TestStand 2.0.1 Operator Interfaces (Old) use this approach. While these examples contain a large amount of complex source code, they provide less

functionality than the simpler examples that use the ActiveX controls. Therefore, National Instruments does not recommend these older examples as a basis for new development.

For more information about writing an application by directly calling the TestStand Engine API, refer to the *Writing an Application with the TestStand Engine API* section of the *TestStand Help*.

Deploying an Operator Interface

Refer to Chapter 14, *Deploying TestStand Systems,* for more information about deploying a TestStand Operator Interface application.

Writing an Application with the TestStand UI Controls

TestStand provides a number of controls that work together to simplify programming an operator interface. These controls fall into two categories—*manager controls* and *visible controls*.

Manager Controls

Manager controls call the TestStand API to perform tasks such as loading files, launching executions, and retrieving sequence 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 that they are viewing. These controls are visible at design time but invisible at run time.

Connect the manager controls to visible TestStand UI Controls to automatically display information or allow the user to select items to view. The following sections describe the specific functionality of the three types of manager controls—Application Manager, SequenceFileView Manager, and ExecutionView Manager.

Application Manager

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

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

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

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

SequenceFileView Manager

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

- Designates a sequence file as the selected sequence file.
- Tracks which sequence, step group, and steps are selected in the selected file.
- Displays aspects of the selected file in the visible TestStand UI controls to which it connects.
- Enables visible TestStand UI Controls to which it connects to change the selected file, sequence, step group, and steps.
- Provides methods for executing the selected sequence file.

Your application needs one SequenceFileView Manager control for each location, such as a window, form, or panel, in which you either display a sequence file or let the user select a current sequence file.

ExecutionView Manager

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

- Designates an execution as the selected execution.
- Tracks which thread, stack frame, sequence, step group, and steps are selected in the selected execution.
- Displays aspects of the selected execution in the visible TestStand UI controls to which it connects.
- Enables visible TestStand UI controls to which it connects to change the selected thread, stack frame, sequence, step group, and steps.
- Sends events to notify your application of the progress and state of the selected execution.

- Provides debugging commands.
- Updates the ReportView control to show the current report for the selected execution.

Your application needs one ExecutionView Manager control for each location, such as a window, form, or panel, in which you either display an execution or let the user select a current execution.

Visible TestStand UI Controls

Visible TestStand UI Controls are visible at both design time and run time. These controls are similar to common Windows UI controls, but they connect to manager controls to automatically display information or to enable the user to select items to view. Table 9-1 describes the visible TestStand UI Controls.

Table 9-1. Visible TestStand UI Controls

Control Name	Description
Button	Connect a manager control to a Button control to specify that the button performs a common operator interface command such as "Open Sequence File". The Button control automatically enables or disables according to the application state. The Button control displays a localized caption.
Label	Connect a manager control to a Label control to automatically display text information about the application state in the label, such as the name of the current user or the status of the current UUT.
StatusBar	Connect a manager control to panes of a StatusBar control to display textual, image, or progress information about the application state. You can also programmatically control individual StatusBar panes to display custom information.
SequenceView	Connect a SequenceFileView Manager control or an ExecutionView Manager control to a SequenceView control to display the steps of a sequence from a selected file or execution. The SequenceView control displays the steps in a list with columns whose contents you specify when you configure the control.

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

Control Name	Description
ComboBox	Connect a SequenceFileView Manager control or an ExecutionView Manager control to a ComboBox control to view or select a list of files, sequences, step groups, executions, threads, or stack frames.
ListBox	Connect a SequenceFileView Manager control or an ExecutionView Manager control to a ListBox control to view or select from a list of files, sequences, step groups, executions, threads, or stack frames.
ListBar	A ListBar control displays multiple pages where each page contains a list of items. You can view and select items from the selected page. Connect a SequenceFileView Manager control or an ExecutionView Manager control to a ListBar page to view and select from a list of files, sequences, step groups, executions, threads, or stack frames.
ReportView	Connect an ExecutionView Manager control to a ReportView control to display the report for the selected execution.
ExpressionEdit	An ExpressionEdit control enables the user to edit a TestStand expression with the convenience of syntax coloring, popup help, and statement completion. While you do not typically need to edit expressions in an operator interface application, you can connect a manager control to a read-only ExpressionEdit control to automatically display text information about the application state, such as the pathname of the selected sequence file or the name of the current user. You can also use ExpressionEdit controls on dialog boxes for step types and tools in which you prompt the user to enter a TestStand expression.

Connecting Manager Controls to Visible Controls

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

The specific connections you can make depend on the type of manager control and visible control that you are connecting. The following kinds of connections are available: view connections, list connections, command connections, and information source connections.

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

View Connections

You can connect manager controls to the SequenceView and ReportView controls to display the current sequence or report, respectively.

Connect a SequenceFileView Manager control to a SequenceView control to display the steps in the selected step group of the selected sequence in the selected file. Connect an ExecutionView Manager control to a SequenceView control to display the steps in the currently executing sequence of the selected execution.

Connect an ExecutionView Manager control to a ReportView control to display the report for the selected execution.

Call the following methods to connect to view controls:

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

List Connections

You can connect a TestStand ComboBox or ListBox control or a ListBar page to a list provided by a manager control. Table 9-2 specifies the available list connections.

Table 9-2. Available List Connections

List	Manager Control
Sequence Files	SequenceFileView Manager
Sequences	SequenceFileView Manager
Step Groups	SequenceFileView Manager
Executions	ExecutionView Manager
Threads	ExecutionView Manager
Stack Frames	ExecutionView Manager

A manager control designates one item in each list as the selected item. A visible control that you connect to a list displays the list and indicates the selected item. The visible control also allows you to change the selected item, unless the application state or control configuration prohibits changing the selection. When you change the selected item, 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 its sequence file list to a combo box. When you change the selected file 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 or ListBox control or a ListBar page:

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

Command Connections

TestStand applications typically provide commands to the user through menus, buttons, or other controls. Many commands are common to most applications, such as OpenSequenceFile, ExecuteEntryPoint, RunSelectedSteps, Break, Resume, Terminate, and Exit. The TestStand UI Controls Library provides a set of common commands you can add to your application. You can connect these commands to TestStand buttons or application menu items. When you connect a command to a button or menu item, the button or menu item automatically executes 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 Library implements many common application commands, connecting commands to buttons and menu items significantly reduces the amount of source code your application requires.

Tip The CommandKinds enumeration defines the set of available commands. Refer to the *TestStand Help* for more information about this enumeration before adding commands to your application so that you do not unnecessarily re-implement an existing command.

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

Refer to the *TestStand Help* for information about each CommandKinds enumeration constant and the manager controls with which the command functions.

Call the following methods to connect a command to a Button control:

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

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

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

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

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

Information Source Connections

Manager controls can connect information sources to Label and ExpressionEdit controls and StatusBar panes to display information about the state of the application. The types of information connections you can establish are caption connections, image connections, and 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 automatically 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 that an ExecutionView Manager control selects. Other captions, such as UserName, function the same regardless of which manager control you use to connect them.

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

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

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

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

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

Image Connections

Image connections display icons that illustrate the status of the application. For example, you can use the ExecutionView Manager control to connect an image to a StatusBar pane so that the pane automatically 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. Images may depend on 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, or it displays an image for the currently executing step group when you connect it to an ExecutionView Manager control.

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

Call the following methods to connect an image to a StatusBar pane:

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

To obtain an image without connecting it to a control, call the following methods:

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

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

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 automatically displays a progress bar that indicates the percentage of progress made in the current execution.

The NumericSources enumeration defines the set of values to which you can connect. Refer to the *TestStand Help* for information about each NumericSources enumeration constant and the manager controls with which the command functions.

To connect a numeric source to a StatusBar pane, call `ExecutionViewMgr.ConnectNumeric`. To obtain a numeric value without connecting it to a control, call `ExecutionViewMgr.GetNumeric`.

Specifying and Changing Control Connections

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

Connections from manager controls to visible controls are many-to-one. Therefore, you can make the same connection from a manager control to multiple visible controls. For example, if you connect two combo boxes to the sequence list of a SequenceFileView Manager control, both combo boxes display the selected sequence in the current file. If you change the selection in one combo box, 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 is already connected, the new connection replaces the existing connection to the visible control.

Using TestStand UI Controls in Different Environments

The following sections describe how to use the TestStand UI Controls in different environments.

LabVIEW

To use the TestStand UI Controls in LabVIEW, use the VIs and UI Controls in the TestStand palette. Refer to Chapter 6, *Creating Custom User Interfaces in LabVIEW*, of *Using LabVIEW with TestStand* for more information about using the TestStand UI Controls in LabVIEW.

LabWindows/CVI

To use the TestStand UI Controls with LabWindows/CVI, add the following files to your project from the `<TestStand>\API\CVI` directory:

- `tsui.fp`—ActiveX API for the TestStand UI Controls
- `tsuisupp.fp`—ActiveX API for use with less commonly used interfaces provided by the TestStand UI Controls
- `tsutil.fp`—Functions that facilitate using the TestStand API and the TestStand UI Controls with LabWindows/CVI
- `tsapicvi.fp`—ActiveX API for the TestStand Engine

Include the following header files, located in the `<TestStand>\API\CVI` directory, in your source files as needed:

- `tsui.h`
- `tsuisupp.h`
- `tsutil.h`
- `tsapicvi.h`.

To add a TestStand UI Control to a panel in the LabWindows/CVI UIR editor, select **ActiveX** from the **Create** menu and select a control that has a name beginning with `TestStand UI`.

Refer to Chapter 6, *Creating Custom User Interfaces in LabWindows/CVI*, of *Using LabWindows/CVI with TestStand* for more information about using the TestStand UI Controls in LabWindows/CVI.

Visual Studio .NET

To use the TestStand UI Controls with Microsoft Visual Studio .NET, drag the TestStand UI Controls from the TestStand tab on the Visual Studio Toolbox onto your form. You must also add a reference to the TSUtil assembly as described in Table 9-6.

Note If the TestStand tab is not visible in the Visual Studio Toolbox window when you edit your form, follow these steps to install it. First, exit all running copies of Visual Studio .NET. Then, run the TestStand Version Switcher utility. Select the current version of TestStand and click **Make Active**.

Visual C++

To use the TestStand UI Controls with Visual C++, add the TestStand Utility (TSUtil) Functions Library to your project as described in the *TestStand Utility Functions Library* section of this chapter. The `TSUtilCPP.cpp` and `TSUtilCPP.h` files automatically import the type libraries for the TestStand API and the TestStand UI Controls.

You can view the header files that the #import directive generates for the TestStand API type libraries by opening the `tsui.tlh`, `tsuisupp.tlh`, and `tsapi.tlh` files that Visual C++ creates in your `Debug` or `Release` directory. These header files define a C++ class for each object class in the TestStand API. The letter `I` is used as a prefix in class names for ActiveX controls and objects that you can create without calling another class.

The header files use macros to define a corresponding smart pointer class for each object class. Each smart pointer class uses the name of its corresponding class and adds a `Ptr` suffix. Typically, you only use smart pointer classes in your application. For example, instead of using the `SequenceFile` class, use the `SequenceFilePtr` class.

Note National Instruments recommends that you use the classes that the #import directive generates to call the TestStand ActiveX API instead of generating MFC wrapper class files using the Class Wizard tool.

To add a TestStand UI Control to a dialog box as a resource, select **Insert ActiveX Control** from the dialog box context menu and select a control whose name begins with `TestStand UI`.

Handling Events

TestStand UI Controls send events to notify your application of user input and of application events, such as an execution completing. The visible controls send user input events such as KeyDown or MouseDown. The manager controls send application state events such as SequenceFileOpened or UserChanged. You can choose to handle any number of events according to the needs of your application.

Creating Event Handlers In Your ADE

Table 9-3 describes how to create an event handler in your specific ADE.

Table 9-3. Creating an Event Handler in Your ADE

ADE	Description
LabVIEW	Register event handler VIs with the **Register Event Callback** node, which is located in the ActiveX palette. Refer to Chapter 6, *Creating Custom User Interfaces in LabVIEW*, of *Using LabVIEW with TestStand* for information about handling events from the TestStand UI Controls in LabVIEW.
LabWindows/CVI	Install ActiveX event callback functions by calling the `TSUI_<object class>EventsRegOn<event name>` functions in `tsui.fp`. Refer to Chapter 6, *Creating Custom User Interfaces in LabWindows/CVI*, of *Using LabWindows/CVI with TestStand* for information about handling events from the TestStand UI Controls in LabWindows/CVI.
.NET	Create .NET control event handlers from the form designer.
C++ (MFC)	Create ActiveX event handlers from the Message Maps page of the Class Wizard dialog box.

Events Handled By Typical Applications

When you create your application, you can direct your application to handle any subset of the available TestStand UI Control events. However, an application typically handles the following events—ExitApplication, Wait, ReportError, DisplaySequenceFile, and DisplayExecution.

ExitApplication

The Application Manager control sends this event to request that your application exit. Handle this event by directing your application to exit normally. For more information about shutting down your application, refer to the *Startup and Shut Down* section of this chapter.

Wait

The Application Manager control sends this event to request that your application either 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 sends this event to request that the operator interface displays to the user an error that occurs 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.

Note This event indicates an application error, not a sequence execution error. The BreakOnRunTimeError event indicates a sequence execution error.

DisplaySequenceFile

The Application Manager control sends this event to request that the application display a particular sequence file. For example, the Application Manager control sends this event when the user opens a sequence file. To handle this event, display the file by setting the `SequenceFileViewMgr.SequenceFile` property. If your application only has a single window, call this method on the SequenceFileView Manager control that resides on that window.

If your application displays each file in a separate window using separate SequenceFileView Manager controls, call `ApplicationMgr.GetSequenceFileViewMgr` to find the SequenceFileView Manager control that currently displays the file so that you can activate the window that contains it. If no SequenceFileView Manager control currently displays the file, a multiple window application can create a new window that contains a SequenceFileView Manager control. The application can then set SequenceFileViewMgr.SequenceFile property of the control to display the file in the new window.

DisplayExecution

The Application Manager control sends this event to request the application to display a particular execution. For example, the Application Manager control sends this event when the user starts a new execution. To handle this event, display the execution by setting the `ExecutionViewMgr.Execution` property. If your application only has a single window, call this method on the ExecutionView Manager control that resides in that window.

If your 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 that you can activate the window that contains it. If no ExecutionView Manager control currently displays the execution, a multiple window application typically creates a new window that contains an ExecutionView Manager control and sets the ExecutionViewMgr.Execution property of the control to display the execution in the new window.

Startup and Shut Down

As a final step in the initialization of your application, call the `ApplicationMgr.Start` method. This method initializes the Application Manager control and launches the LoginLogout Front-End callback if you have not set the `ApplicationMgr.LoginOnStart` property to `False`.

Complete the following steps to shut down your application:

1. If your application holds any references to TestStand objects such as sequence files or executions, handle the `ApplicationMgr.QueryShutDown` event. To respond to the event, cancel the shut down process or release the TestStand object references your application holds.
2. Call `ApplicationMgr.ShutDown`. If this method returns `True`, exit your application. If the method returns `False`, do not exit the application. Leaving the application running allows the method to shut down any running executions and unload sequence files. If the shut down process completes, the Application Manager control sends the ExitApplication event to notify you that you can now exit the application. If the shut down process is cancelled, the Application Manager control sends the ShutDownCancelled event. This occurs if the user chooses not to terminate a busy execution.

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 ExitApplication event.

Note When you use the TestStand UI Controls to create an Exit button or an Exit menu item that invokes the Exit command, the button or menu item automatically calls ApplicationMgr.ShutDown for you.

TestStand Utility Functions Library

The TestStand Utility (TSUtil) Functions Library is a set of functions that help you to use certain aspects of the TestStand API in particular programming environments. Many TSUtil functions operate on environment-specific objects, such as menus, that the environment-neutral TestStand API cannot access. The functions available in TSUtil vary according to your programming environment.

To assist operator interface developers, the TSUtil library contains functions to insert menu items that automatically execute commands that the TestStand UI Controls library provides. The TSUtil library also provides functions that help you localize the strings on your user interface.

Refer to the *Menus and Menu Items* section of this chapter for information about using TSUtil functions to create menu items that perform common TestStand commands. Refer to the *Localization* section of this chapter for information about how to display application user interface strings in a selected language.

The following tables describe how to use the TSUtil library in each programming environment:

Table 9-4. Using the TSUtil Library in LabVIEW

Library Format	VIs in the block diagram palette for TestStand.
Help Location	VI help inside each VI.
How to Use	Insert VIs on the block diagram as needed. Refer to *Using LabVIEW with TestStand* for more information about using the TSUtil library in LabVIEW.

Table 9-5. Using the TSUtil Library in LabWindows/CVI

Library Format	Instrument Driver.
Help Location	Function panels (`TSUtil.fp`).
Files	`TSUtil.c`, `TSUtil.h`, `TSUtil.fp`, and `TSUtil.obj` (located in the `<TestStand>\API\CVI` directory).
How to Use	Insert `TSUtil.fp` into your LabWindows/CVI project. Include `TSUtil.h` in your source files as needed. The names of TestStand-related functions in this library begin with a `TS_` prefix.

Table 9-6. Using the TSUtil Library in .NET Languages

Library Format	Assembly.
Help Location	In the Object Browser and in the source window using Intellisense.
Location	`<TestStand>\API\DotNet\Assemblies\CurrentVersion.`
File	`NationalInstruments.TestStand.Utility.dll.`
How to Use	Add a reference to the assembly to your project. The classes in this assembly reside in the `National Instruments.TestStand.Utility` namespace. To add a reference to the assembly, select **Add Reference** from the **Project** menu in Visual Studio .NET to launch the Add Reference dialog box. Next, select the **.NET** tab and click the **File Browse** button to launch the Select Components dialog box. Then, navigate to the `<TestStand>\API\DotNet\Assemblies\CurrentVersion` directory. Select `NationalInstruments.TestStand.Utility.dll`, and click **Open**. Click **OK** to close the Add Reference dialog box.

Table 9-7. Using the TSUtil Library in C++ (MFC)

Library Format	C++ source code.
Help Location	Comments in the C++ header file, `TSUtilCPP.h`.
Files	`TSUtilCPP.cpp` and `TSUtilCPP.h` (located in the `<TestStand>\API\VC` directory).
How to Use	Add `TSUtilCPP.cpp` to your project once. Include `TSUtilCPP.h` in your source files as needed. The classes in this library reside in the `TSUtil` namespace.

If your programming environment is not described in this section, there is not a version of TSUtil for your environment. In this case, you can write your own code that performs equivalently to any functions you need from the TSUtil library. You can use the source code for one of the existing TSUtil libraries as a guide.

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 handle events for menu items. You can greatly reduce the amount of code required to implement menus in your application by calling TSUtil functions to create menu items that invoke TestStand commands. These menu items are automatically dimmed or enabled according to the application state and set their captions according to the selected language. The menu items execute their commands automatically so that your application does not need to handle menu events or provide command implementations.

Your 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 your project as described in the *TestStand Utility Functions Library* section of this chapter.

Updating Menus

The contents of a menu can vary, depending on the current selection or other user input, or due to asynchronous execution state changes. Instead of updating a menu in response to any event or change that may affect it, it is simpler to update the state of a menu just before it displays when the user opens it. Programming environments provide an event that notifies you when a menu is about to open. Table 9-8 describes the notification method for each environment.

Table 9-8. Menu Open Notification Methods by ADE

Environment	Menu Open Notification Method
LabVIEW	`<This VI>:Menu Activation?` event Refer to *Using LabVIEW with TestStand* for information about how to determine when a menu is about to open in LabVIEW.
LabWindows/CVI	`InstallMenuDimmerCallback` Refer to *Using LabWindows/CVI with TestStand* for information about how to determine when a menu is about to open in LabWindows/CVI.
.NET	`Form.MenuStart`
C++ (MFC)	`CWnd::OnInitMenuPopup`

To handle this notification, use the following TSUtil functions to remove and reinsert all TestStand menu items from your menus: `RemoveMenuCommands`, `InsertCommandsInMenu`, and `CleanupMenu`.

InsertCommandsInMenu takes 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.

Note You can insert and remove TestStand commands in menus that also contain non-TestStand menu items.

Refer to the *TestStand Utility Functions Library* section of this chapter for the full set of menu support functions specific to your environment and

descriptions of how to use them. Refer to the examples in the `<TestStand>\OperatorInterfaces\NI\Full-Featured` directory for sample code that handles open notification events for menus.

Localization

The Engine.StationOptions.Language property specifies the current language. Localized TestStand applications use the Engine.GetResourceString method to obtain text in the current system language from language resource files. Refer to the *Creating String Resource Files* section of Chapter 8, *Customizing and Configuring TestStand*, for information about creating your own string resource files.

To localize all of the user-visible TestStand UI Control strings that you configure at design time, call `ApplicationMgr.LocalizeAllControls`. This reduces the number of strings you must explicitly localize using Engine.GetResourceString 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.

Note Buttons and menu items that you connect to commands automatically localize their caption text. Refer to the *Command Connections* section of this chapter for more information about connecting buttons and menu items to commands.

The LocalizeAllControls method operates on TestStand UI Controls only. For other controls and user interface elements, your application must set each item of localized text. However, the TSUtil library provides functions to assist in localizing these other controls and menu items. These functions are described in Table 9-9.

Table 9-9. TSUtil Library Localization Functions by Environment

Environment	TSUtil Library Localization Function

LabVIEW	`TestStand - Localize Front Panel.vi`
	`TestStand - Localize Menu.vi`
	`TestStand - Get Resource String.vi`

LabWindows/CVI	`TS_LoadPanelResourceStrings`
	`TS_LoadMenuBarResourceStrings`
	`TS_SetAttrFromResourceString`
	`TS_GetResourceString`

Table 9-9. TSUtil Library Localization Functions by Environment (Continued)

Environment	TSUtil Library Localization Function
.NET	`Localizer.LocalizeForm` `Localizer.LocalizeMenu`

C++ (MFC)	`Localizer.LocalizeWindow`
	`Localizer.LocalizeMenu`
	`Localizer.LocalizeString`

For more information about the TSUtil library, refer to the *TestStand Utility Functions Library* section of this chapter.

Operator Interface Application Styles

Although you can use the TestStand UI Controls to create any type of application, the following application formats are the most common: single window, multiple window, or no visible window. Applications of a particular style tend to share a similar implementation strategy, particularly with respect to their use of the TestStand manager controls. The following sections describe implementation strategies for these common application styles.

Tip Since the structure of your application may not exactly match one of the described applications, only use these descriptions as a guide. You should implement the approach that best suits your application.

Single Window

A single window application typically displays one execution and/or sequence file at a time. The user can select the execution and sequence file to display from a ListBar, ComboBox, or ListBox control. The examples in the `<TestStand>\OperatorInterfaces\NI\Full-Featured` and `<TestStand>\OperatorInterfaces\NI\Simple` directories are single window applications.

The single window application contains one Application Manager control, one SequenceFileView Manager control, and one ExecutionView Manager control. To display sequences, you can 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 ExecutionView Manager control connect to separate SequenceView controls, but only one SequenceView control is visible at a time. Visibility is based on whether you select to view sequence files or executions in the list bar.

In the examples in the `Simple` directory, the ExecutionView Manager control connects to the SequenceView control. Since the SequenceFileView Manager control does not connect to a SequenceView control, these examples only display sequences for the current execution and do not display sequences from the selected sequence file.

Multiple Window

A multiple window application has at least one window that always exists in order to contain the Application Manager control. While this window may be visible or invisible, it is typically visible and contains controls that enable the user to open sequence files.

For each sequence file that you open, the application creates a Sequence File window that contains a SequenceFileView Manager control and a SequenceView control to which it 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, it calls ApplicationMgr.GetSequenceFileViewMgr to determine whether a SequenceFileView Manager control is currently displaying the sequence file. If so, the application retrieves the window from the SequenceFileViewMgr.UserData property and activates the window. If there is no window currently displaying the file, the application creates a new window and sets the SequenceFileViewMgr.SequenceFile property to display the specified file. Because the window only displays this file, the application also sets the SequenceFileViewMgr.ReplaceSequenceFileOnClose property to `False`.

If a Sequence File window attempts to close and the SequenceFileViewMgr.SequenceFile property is `NULL`, the application allows the window to close immediately. If the SequenceFileViewMgr.Sequence File property is not `NULL`, the application does not close the window. Instead, the application passes the file to the ApplicationMgr.CloseSequenceFile method. When the application receives the SequenceFileViewMgr.SequenceFileChanged event with a

NULL sequence file event parameter, it closes the window that holds the SequenceFileView Manager control.

The Sequence File window contains controls that allow you to execute the displayed file. For each execution that you start, the application creates an Execution window that contains an ExecutionView Manager control and a SequenceView control to which it 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, it calls ApplicationMgr.GetExecutionViewMgr to determine whether an ExecutionView Manager control is currently displaying the execution. If so, the application retrieves the window from the ExecutionViewMgr.UserData property and activates the window. If there is no window currently displaying the execution, the application creates a new window and sets the ExecutionViewMgr.Execution property to display the specified execution. Because the window only displays this execution, the application also sets the ExecutionViewMgr.ReplaceExecutionOnClose property to False.

If an Execution window attempts to close and the ExecutionViewMgr.Execution property is NULL, the application allows the window to close immediately. If the ExecutionViewMgr.Execution property is not NULL, the application does not close the window and instead passes the execution to the ApplicationMgr.CloseExecution method. The application does not immediately close the Execution window to ensure that it exists until the execution it displays completes. When the application receives the ExecutionViewMgr.ExecutionChanged event with a null execution event parameter, it closes the window that holds the ExecutionView Manager control.

A multiple window application can display multiple child windows instead of displaying sequence files and executions in separate top-level windows. Child windows can be simultaneously visible or reside in tab control pages or similar locations that allow you to easily select which child window to view.

No Visible Window

An application without a visible window is similar to a single window application except that it does not display its window. The application can allow its command-line arguments to execute and then exit, or it might have a different mechanism to determine which files to load and execute. Although an invisible application does not require an ExecutionView Manager control, it may use a SequenceFileView Manager control to provide methods to launch an execution for a selected file. Use the following SequenceFileView Manager control properties and methods to launch executions:

- `ExecutionEntryPoints`
- `Run`
- `RunSelectedSteps`
- `LoopOnSelectedSteps`
- `GetCommand`

Command-Line Arguments

The Application Manager control automatically processes the command line that invokes your application when you call ApplicationMgr.Start. To disable command-line processing, set the `ApplicationMgr.ProcessCommandLine` property to `False` before calling `ApplicationMgr.Start`. Refer to the *Configuring TestStand* section of Chapter 8, *Customizing and Configuring TestStand*, for a description of the command-line arguments that are processed by the Application Manager control.

Your application can also process the command line to support additional command-line flags. The ApplicationMgr.CommandLineArguments property provides the command-line arguments with which your application was invoked.

Note Any command-line flags that you add to your application must appear after any TestStand command-line flags because the Application Manager control will stop processing the command line when it encounters a flag it does not recognize.

Persistence of Application Settings

The TestStand Engine stores Station Option dialog box settings and other settings that apply to all TestStand applications. However, each operator interface also stores additional custom settings. These settings include items such as whether to break on the first step of execution, whether to break when a step fails, and the list of most recently used sequence files. The Application Manager control stores these settings in the configuration file specified by the ApplicationMgr.ConfigFilePath property.

Configuration File Location

The default location of the ApplicationMgr.ConfigFilePath property is `%UserProfile%\Local Settings\Application Data\OperatorInterface.ini`. This path specifies a directory to which the Windows user who is currently logged in has permission to write files. To change the configuration file location, set the `ApplicationMgr.ConfigFilePath` property before your application calls `ApplicationMgr.Start`.

If you specify a relative file path or just a file name, that file location is relative to the directory that contains your application. If other users who do not have Windows administrator privileges can run your application, you must ensure that your configuration file is stored in a location to which your users have permission to write files.

Adding Custom Application Settings

After your 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 `Data` property of this property object file to obtain the PropertyObject that holds the application settings.
3. Ensure your custom setting exists in this PropertyObject by setting a default value.

   To set the default value of your setting, call a method such as `PropertyObject.SetValBoolean` with a lookup string such as `"CustomSettings.MyExampleBooleanSetting"` and an options parameter of `PropOption_SetOnlyIfDoesNotExist`.

4. Call a method such as `PropertyObject.SetValBoolean` with an options parameter of `PropOption_NoOptions` to set your custom option in response to user input.
5. Call a method such as `PropertyObject.GetValBoolean` to obtain the current value of your custom option.

When you call ApplicationMgr.ShutDown or change any Application Manager control setting, the Application Manager control persists the application settings to the configuration file. You can also persist the settings at anytime by calling `PropertyObjectFile.WriteFile`.

Using the TestStand API With TestStand UI Controls

The TestStand UI Controls greatly reduce the need for an application to directly call the TestStand API. However, you can still call the TestStand API directly on objects you create or obtain from the TestStand UI Controls methods, properties, or events. Note the following guidelines when you call the TestStand API in an operator interface that uses the TestStand UI Controls:

- You do not need to create the TestStand Engine. Instead, you can obtain the Engine object using the `ApplicationMgr.GetEngine` method.
- If you create an execution by calling `Engine.NewExecution`, the TestStand UI Controls recognize the new execution.
- If you load a sequence file by calling `Engine.GetSequenceFileEx`, the TestStand UI Controls are not aware of the file you load. To open and display a file in the operator interface, you must call `ApplicationMgr.OpenSequenceFile`.
- You can obtain sequence file and execution references from events or from the SequenceFiles and Executions collections.
- If you hold references to TestStand objects, release them in the handler for the `ApplicationMgr.QueryShutdown` event if your event handler does not cancel the shut down process.

10

Customizing Process Models and Callbacks

This chapter describes how to customize the TestStand process models and callbacks. For detailed information about the TestStand process models, refer to Appendix A, *Process Model Architecture*.

Process Models

All process models that TestStand provides identify UUTs, generate test reports, log results to databases, and display UUT status. These process models also allow client sequence files to customize various model operations by overriding model-defined callback sequences.

Process models provide Configuration and Execution entry points that you use to configure model settings and to run client files under the model. These entry points are sequences in the process model sequence file and are typically listed in the Configure and Execute menus of an application.

The models that TestStand provides—Sequential, Parallel, and Batch—contain the following Execution entry points:

- **Test UUTs**—Tests and identifies multiple UUTs or UUT batches in a loop.
- **Single Pass**—Tests one UUT or a single batch of UUTs without identifying them.

The TestStand process models also contain the following Configuration entry points:

- **Report Options**—Launches the Report Options dialog box, in which you can configure the location and contents of report files.
- **Database Options**—Launches the Database Options dialog box, in which you can configure the location and contents of databases.
- **Model Options**—Launches the Model Options dialog box, in which you can configure the number of test sockets and other options related to process models.

Station Model

You can specify a process model file to use for all sequence files. This process model file is called the *station model file*. The Sequential model is the default station model file. You can use the Station Options dialog box to select a different station model or to allow individual sequence files to specify their own process model file.

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

Specifying a Specific Process Model for a Sequence File

If TestStand is configured to allow individual sequence files to specify their own process model files, you can set the process model file of a sequence file in the Sequence File Properties dialog box. You can also specify that a sequence file does not use a process model.

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

Modifying the Process Model

To make changes to the process model that apply wherever the process model is used, you must modify the process model directly.

TestStand installs the process model sequence files—`SequentialModel.seq`, `ParallelModel.seq`, and `BatchModel.seq`—and their supporting files to the `<TestStand>\Components\NI\Models\TestStandModels` directory.

If you want to change or enhance the process model files, copy the entire contents of the `<TestStand>\Components\NI\Models\TestStandModels` directory to `<TestStand>\Components\User\Models` and make changes to this copy. This practice ensures that newer installations of TestStand do not overwrite your customizations.

Tip Remember that process models are TestStand sequence files. To modify the behavior of process models, edit them for desired functionality as you would any other sequence files. For example, if you want to change the HTML report output for all sequences, copy `reportgen_html.seq` from the `NI` directory to the `User` directory and then make changes to that copy.

Process Model Callbacks

Model callbacks allow you to customize the behavior of a process model for each client sequence file that uses it. By defining one or more Model callbacks in a process model, you can specify the process model operations that you can customize from your client sequence file.

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

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

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

Special Editing Capabilities for Process Model Sequence Files

The TestStand Sequence Editor has specific features for creating or modifying process model sequence files.

If you want TestStand to treat a sequence file as a process model, you must mark it as a process model file. To do so, select **Edit»Sequence File Properties**. In the Sequence File Properties dialog box, select the **Advanced** tab, and then select **Model** from the Type ring control.

Figure 10-1 shows the settings on the Advanced tab on the Sequence File Properties dialog box.

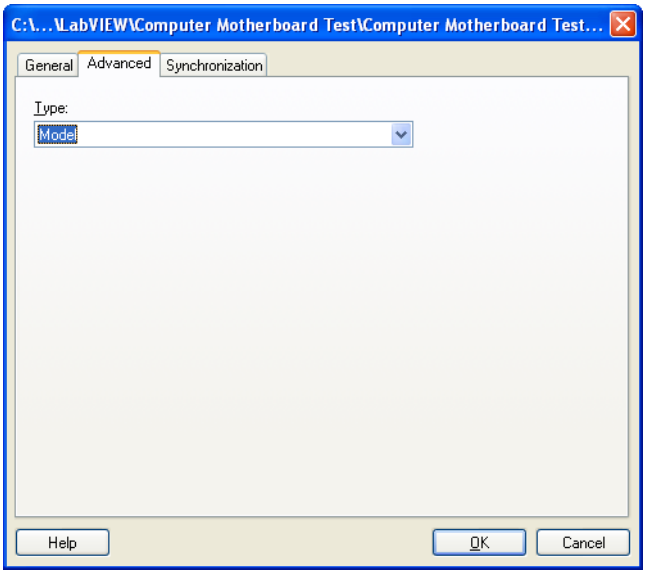


Figure 10-1. Sequence File Type Setting on the Advanced tab on the Sequence File Properties Dialog Box

Although you edit a process model sequence file in a regular Sequence File window, the file has special contents. In particular, some of the sequences in process model files are Model entry points, and some are Model callbacks. TestStand maintains special properties for entry point and callback sequences. You can specify the values of these properties when you edit the sequences in a process model file. When you access the Sequence Properties dialog box for any sequence in a model file, it contains a Model tab that allows you to specify whether the sequence is a normal callback or an entry point sequence.

Normal Sequences

A normal sequence is any sequence other than a callback or entry point. In a process model file, you use normal sequences as Utility subsequences that the entry points or callbacks call. When you select **Normal** from the Type ring control, nothing else is listed on the Model tab.

Callback Sequences

Model callbacks are sequences that entry point sequences call and that the client sequence file can override. By marking sequences as callbacks in a process model file, you specify the set of process model operations that you can customize. When editing the client file, select **Edit»Sequence File Callbacks** to override the callback. Refer to the *TestStand Help* for more information about using the Sequence File Callbacks dialog box.

Some Model callbacks, such as the TestReport callback in the default process model, have implementations sufficient for handling most types of test results. Other Model callbacks act as placeholders that you can override with sequences in the client sequence file. For example, the MainSequence callback in the model file is a placeholder for the MainSequence callback in the client sequence file.

Entry Point Sequences

Entry point sequences are sequences that you can invoke from the menus in the TestStand Sequence Editor or Operator Interfaces. You can specify two different types of entry points: Execution entry points and Configuration entry points.

- **Execution entry point**—Runs test programs. Execution entry points typically call the MainSequence callback in the client sequence file. The default process model contains two Execution entry points: Test UUTs and Single Pass. By default, Execution entry points are listed in the Execute menu. Execution entry points are only listed in the menu when the active window contains a sequence file that has a MainSequence callback.
- **Configuration entry point**—Configures a feature of the process model. Configuration entry points usually save the configuration information in a `.ini` file in the `<TestStand>\Cfg` directory. By default, Configuration entry points are listed in the Configure menu. The default process model contains the Configuration entry point, Configure Report Options. The Configure Report Options entry point is listed as Report Options in the Configure menu.

Callbacks

In addition to the Model callbacks, TestStand includes many other callback sequences that you can customize to meet your specific needs. These callbacks are divided into two groups—Engine callbacks and Front-End callbacks.

Engine Callbacks

The TestStand Engine defines a set of *Engine callbacks* which it invokes at specific points during execution.

Engine callbacks allow you to configure TestStand to call certain sequences at various points during your 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 defines the set of Engine callbacks and their names, since the TestStand Engine controls the execution of steps and the loading and unloading of sequence files.

The Engine callbacks are categorized according to the file in which the callback sequence appears. You can define Engine callbacks in sequence files, process model files, and the `StationCallbacks.seq` file.

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

Note TestStand installs predefined Station Engine callbacks in the `StationCallbacks.seq` file in the `<TestStand>\Components\NI\Callbacks\Station` directory. Add your own Station Engine callbacks in the `StationCallbacks.seq` file in the `<TestStand>\Components\User\Callbacks\Station` directory.

Table 10-1. Engine Callbacks

Engine Callback	Where You Define the Callback	When the Engine Calls the Callback
SequenceFilePreStep	Any sequence file	Before the engine executes each step in the sequence file.
SequenceFilePostStep	Any sequence file	After the engine executes each step in the sequence file.
SequenceFilePreInteractive	Any sequence file	Before the engine begins an interactive execution of steps in the sequence file.
SequenceFilePostInteractive	Any sequence file	After the engine completes an interactive execution of steps in the sequence file.
SequenceFileLoad	Any sequence file	When the engine loads the sequence file into memory.
SequenceFileUnload	Any sequence file	When the engine unloads the sequence file from memory.
SequenceFilePostResultList Entry	Any sequence file	After the engine fills out the step result for a step in the sequence file.
SequenceFilePostStepRuntimeError	Any sequence file	After a step in the sequence file generates a run-time error.
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 that the process model calls, and each step in any resulting subsequence calls.

Table 10-1. Engine Callbacks (Continued)

Engine Callback	Where You Define the Callback	When the Engine Calls the Callback
ProcessModelPostStep	Process model file	After the engine executes each step in any client sequence file that the process model calls, and each step in any resulting subsequence calls.
ProcessModelPreInteractive	Process model file	Before the engine begins interactive execution of steps in a client sequence file.
ProcessModelPostInteractive	Process model file	After the engine begins interactive execution of steps in a client sequence file.
ProcessModelPostResultList Entry	Process model file	After the engine fills out the step result for a step in any client sequence file that the process model calls or in any resulting subsequence calls.
ProcessModelPostStepRuntimeError	Process model file	After a step generates a run-time error when the step is in a client sequence file that the process model calls or in any resulting subsequence calls.
ProcessModelPostStepFailure	Process model file	After a step fails when the step is in a client sequence file that the process model calls or in any resulting subsequence calls.
StationPreStep	`StationCallbacks.seq`	Before the engine executes each step in any sequence file.

Table 10-1. Engine Callbacks (Continued)

Engine Callback	Where You Define the Callback	When the Engine Calls the Callback
StationPostStep	`StationCallbacks.seq`	After the engine executes each step in any sequence file .
StationPreInteractive	`StationCallbacks.seq`	Before the engine begins any interactive execution.
StationPostInteractive	`StationCallbacks.seq`	After the engine completes any interactive execution.
StationPostResultListEntry	`StationCallbacks.seq`	After the engine fills out the step result for a step in any sequence file.
StationPostStepRuntimeError	`StationCallbacks.seq`	After any step generates a run-time error.
StationPostStepFailure	`StationCallbacks.seq`	After any step fails.

The following are examples of how you can use Engine callbacks:

- Use the SequenceFileLoad callback to ensure that the configuration for external devices that the subsequence file uses only occurs once during execution. Usually, you initialize the devices that a sequence requires by creating steps in the Setup step group for the sequence. However, if you call the sequence repeatedly, you can move the Setup steps into a SequenceFileLoad callback for the subsequence file so that they only run 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.

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

Note Do not define a SequenceFileLoad or SequenceFileUnload callback in the `StationCallbacks.seq` sequence file. TestStand does not call these callbacks.

Note If a callback sequence is empty, TestStand does not invoke an Engine callback. Also, the process model uses the Engine.EnableCallback method to disable the ProcessModelPostResultListEntry and SequenceFilePostResultListEntry callbacks when the Discard Results or Disable Results When Not Required by Model setting is enabled on the Model Options dialog box.

Note TestStand only calls other Engine callbacks when executing the SequenceFileLoad and SequenceFileUnload Engine callbacks. TestStand does not call Engine callbacks when executing the other Engine callbacks.

Front-End Callbacks

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

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

You can create new Front-End callbacks by adding a sequence to the `FrontEndCallbacks.seq` file. You can then invoke this sequence from each of your operator interface applications using functions in the TestStand API. However, you cannot edit the source for the TestStand Sequence Editor and therefore cannot make the sequence editor call new Front-End callbacks that you create.

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

11

Type Concepts

This chapter discusses concepts that apply to step types, custom named data types, and standard named data types in TestStand. This chapter also describes the Type Palette window.

For an overview of the categories of types, refer to the *Step Types* and *Standard and Custom Data Types* sections of Chapter 1, *TestStand Architecture*, of this manual.

Creating and Modifying Types

This section describes the windows and views in which you can create, modify, or examine data types and step types. This section also describes how TestStand stores the definitions for data types and step types.

Where You Create and Modify Types

Table 11-1 describes each graphical user interface (GUI) where you can access data types and step types: the windows, the views, the contents of a display, and the corresponding files. Each display presents the types that correspond to the file that you have open.

Table 11-1. GUIs for Accessing Data Types and Step Types

Window	View within the Window	Contents of the Display	Corresponding Files
Sequence File window	Sequence File Types view	Tabs for the step types, custom data types, and standard data types that the variables and steps in the sequence file use.	When you save the contents of the Sequence File window, TestStand writes the definitions of the types used in the sequence file to the sequence file. Refer to the *TestStand Help* for more information about the Sequence File window.

Table 11-1. GUIs for Accessing Data Types and Step Types (Continued)

Window	View within the Window	Contents of the Display	Corresponding Files
Station Globals window	Global Types view	Tabs for the custom data types and standard data types that the station global variables use.	When you save the contents of the Station Globals window, TestStand writes the definitions of the types used in station global variables to the `StationGlobals.ini` file in the `<TestStand>\Cfg` directory. Refer to the *TestStand Help* for more information about the Station Globals window.
User Manager window	Types view	Tabs for the custom data types and standard data types that the User objects use.	All users and user profiles use the User standard data type. To customize the User standard data type, add subproperties to it on the Standard Data Types tab. If any of these subproperties use custom data types, the custom data types appear on the Custom Data Types tab. When you save the contents of the User Manager window, TestStand writes the definitions of the types used to define users and user profiles to the `Users.ini` file in the `<TestStand>\Cfg` directory. Refer to the *TestStand Help* for more information about the User Manager window.
Type Palette window	One view for each type palette file	Tabs for the step types, custom data types, and standard data types that you want to have available in the sequence editor at all times.	By dragging a type into a type palette file in the Type Palette window, you ensure that the type is always available even when it is not in the Types views of the User Manager window, the Station Globals window, or any of the open Sequence File windows. When you save the Type Palette window, TestStand saves all type palette files. Typically, type palette files reside in the `<TestStand>\Cfg\TypePalettes` directory. Refer to the *Type Palette Window* section of this chapter for more information.

Storing Types in Files and Memory

For each type that a TestStand file uses, TestStand stores the definition of the type in the file. You can also specify that a file always saves the definition for a type, even if it does not currently use the type. Because many files can use the same type, many files can contain definitions for the same type. All your sequence files, for example, might contain the definitions for the Pass/Fail Test step type and the CommonResults standard data type.

TestStand only allows one definition for each type in memory. Although the type can appear in multiple views, only one underlying definition of the type exists in memory. If you modify the type in one view, it updates in all views. The Find Type command in the sequence editor's View menu launches the Find Type dialog box, which contains a list of all types that are currently in memory. The list identifies the set of files that use each type.

If you load a file that contains a type definition and another type definition of the same name already exists in memory, TestStand verifies that the two type definitions are identical. If they are not identical, TestStand informs you of the conflict through the Type Conflict In File dialog box, which allows you to resolve the conflict.

Refer to the *TestStand Help* for more information about the Find Type and Type Conflict In File dialog boxes.

Type Palette Window

Use the Type Palette window to view and edit Type Palette files. You use Type Palette files to store the data types and step types that you want to be available in the sequence editor at all times.

When you create a new type in the Sequence File Types view of a Sequence File window, the type does not appear in the Insert Local, Insert Global, Insert Parameter, Insert Field, and Insert Step submenus in other Sequence File windows. To use the type in other sequence files, you can manually copy or drag the new type from one Sequence File window to another. A better approach is to copy or drag the new type to the Type Palette window or to recreate it there. Each type in a Type Palette file appears in the appropriate Insert submenu for all windows.

When you save the contents of the Type Palette window, TestStand writes the contents of all modified type palette files. Typically, type palette files are stored in the `<TestStand>\Cfg\TypePalettes` directory.

You can distribute step types and data types you create to other machines by installing your type palette file to the `<TestStand>\Cfg\TypePalettes` directory. You must prefix the file names 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 type palette file to install whose base file name is not the same as any existing type palette, TestStand removes the Install_ prefix and adds the type palette to the type palette list. When TestStand finds a type palette file to install whose base file name is the same as an existing type palette, TestStand merges the types from the install file into the existing type palette file and then deletes the install file.

The Type Palette window contains tabs that display the step types, custom data types, and standard data types in the selected type palette file. Typically, you create new types in the `MyTypes.ini` type palette file or in a new type palette file that you create.

Refer to the *TestStand Help* for more information about the Type Palette window.

12

Standard and Custom Data Types

This chapter describes how to use data types in TestStand and how to create and modify custom data types to meet the needs of your application.

Using Data Types

You use data types when you insert variables, parameters, or step properties. Each view in which you can insert a variable, parameter, or property features a context menu that contains an Insert item. You can use the context menu items in the views that are listed in Table 12-1.

Table 12-1. Creating Data Type Instances from Context Menus

Context Menu Item	Location of Context Menu	Item Inserted
Insert Global	Sequence File Globals view of the Sequence File window	Sequence file global variable
Insert Parameter	Parameters tab in individual sequence file views in the Sequence File window	Sequence parameter
Insert Local	Locals tab in individual sequence file views in the Sequence File window	Sequence local variable
Insert Global	Globals view of the Station Globals window	Station global variable
Insert User	Users view of the User Manager window	New object with the User data type
Insert Field	Type Palette window and the Types views in the Sequence File, Station Globals, or User Manager windows	New element in an existing data type

With the exception of the Insert User item, all of the context menu items in Table 12-1 provide a submenu from which you can select a data type. The submenu includes the following categories of types:

- One of the simple data types that TestStand defines, including the number, Boolean, string, and object reference data types.
- A named data type. This submenu includes all of the custom named data types that are currently in the Type Palette window or in the Types view of the window you are currently editing. The submenu also includes standard named data types that come with TestStand, such as Error, Path, and CommonResults. Refer to the *Using the Standard Named Data Types* section of this chapter for more information about the standard named data types.
- An array of elements that all have the same data type.

In the submenu for Insert Parameter, you can also select the Container type. You cannot add fields to parameters you create with the Container type. Creating a parameter with the Container type is only useful if you want to pass an object of any type to the sequence. To do so, you must also turn off type checking for the parameter.

To create a parameter with a complex data type, you must first create the data type in the Sequence File Types view or the Type Palette window. Then select the data type from the Types submenu in the Insert Parameter submenu.

Figure 12-1 shows the Insert Local submenu. The submenu includes three custom data types as examples: Fixture, Subassembly, and YieldStatistics.

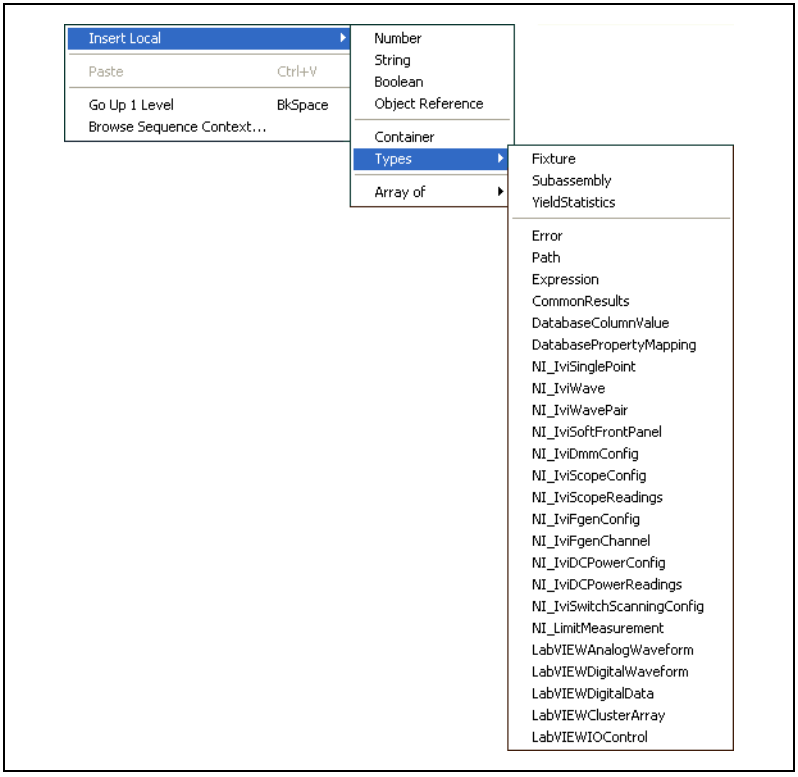


Figure 12-1. Insert Local Submenu

If the submenu does not contain the data type you require, you must create the data type in the Type Palette window or one of the type views. If the data type already exists in another window, drag or copy the data type from the other window to the window you are editing or to the Type Palette window.

Specifying Array Sizes

When you select an item from the Array of submenu in an Insert submenu, the Array Bounds dialog box launches.

Figure 12-2 shows the Array Bounds dialog box with settings for a three-dimensional array.

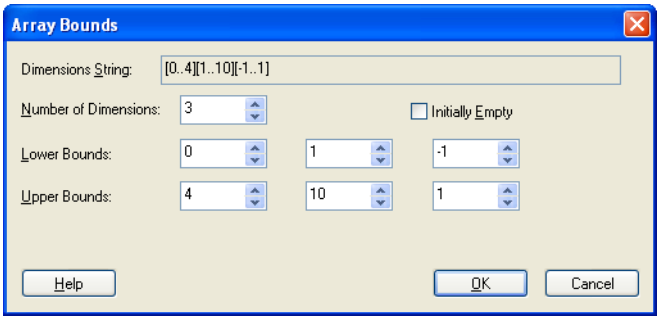


Figure 12-2. Array Bounds Dialog Box

The first and outermost dimension has five elements, with `0` as the minimum index and `4` as the maximum index. The second dimension has ten elements, with `1` as the minimum index and `10` as the maximum index. The third and innermost dimension has three elements, with `-1` as the minimum index and `1` as the maximum index.

After you create a variable, parameter, or property as an array, you can modify the array bounds by selecting the Properties item in the context menu for the variable, parameter, or property in the list view. Select the Bounds tab that is visible in the Properties dialog box to modify the array bounds.

Empty Arrays

If you want the array to be empty when you start the execution, enable the **Initially Empty** option. When you enable this option, the Upper Bounds control for each dimension dims. Defining an array as initially empty is useful if you do not know the maximum array size the sequence requires during execution or if you want to save memory during the periods of execution when the sequence does not use the array.

Figure 12-3 shows the Array Bounds dialog box with settings for a three-dimensional array that is initially empty.

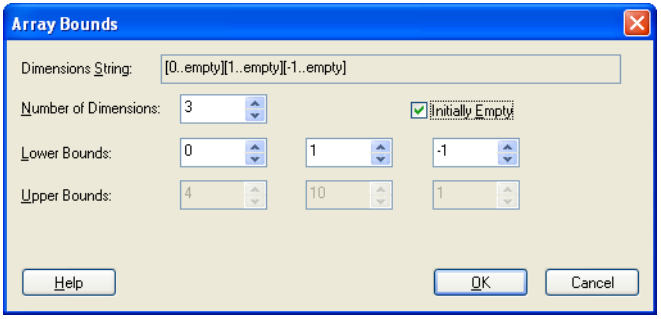


Figure 12-3. Array Bounds Dialog Box with an Initially Empty Array

Display of Data Types

The data type of each variable or property you create is listed in the Type column next to the variable or property name. If the data type is an array, the words `Array of` appear in the Type column, followed by the data type of the array elements and the range of each dimension. If the data type is a named data type, the underlying type is listed in the Type column, followed by the words `Instance of Type` and the data type name.

Figure 12-4 shows variables with different data types on the Locals tab in the Sequence File window. Table 12-2 describes the data type of each local variable in Figure 12-4.

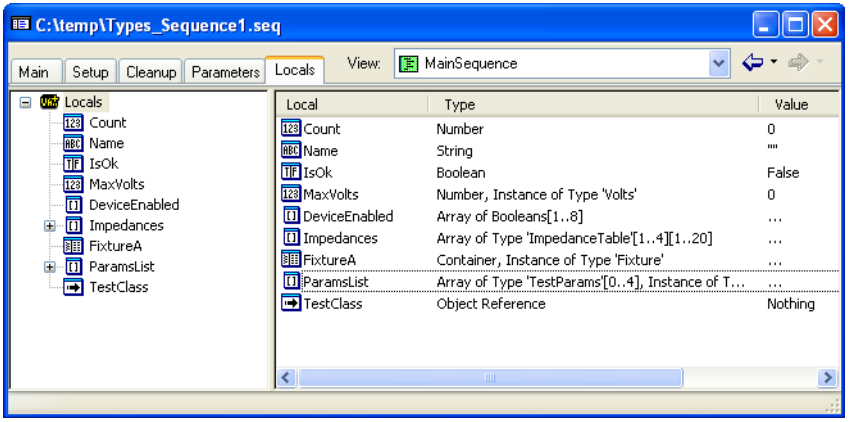


Figure 12-4. Local Variables with Various Data Types

Table 12-2. Data Types of the Local Variables

Local Variable	Data Type	Description
Count	Number	Predefined by TestStand.
Name	String	Predefined by TestStand.
IsOk	Boolean	Predefined by TestStand.
MaxVolts	Volts	Custom data type.
DeviceEnabled	Boolean	One-dimensional array of Booleans, with indexes from 1 to 8.
Impedances	ImpedanceTable	Custom data type that represents a two-dimensional array of numbers.
FixtureA	Fixture	Represents a container that contains multiple fields with different data types.
ParamsList	TestParamList	Represents a one-dimensional array of elements with the TestParams data type. The TestParams data type represents a container that contains multiple fields with different data types.
TestClass	ObjectReference	Predefined by TestStand.

Modifying Data Types and Values

With the exception of resizing arrays, you cannot change the internal structure of a variable, parameter, or property after you create it. You cannot change its data type setting, nor can you deviate from the data type. You can, however, change the contents of the data type itself. Changing the contents of a data type affects all variables, parameters, and properties that use the data type.

You can modify the value of a variable, parameter, or property in the list view in which you create it. For variables and properties, this value is the initial value when you start execution or call the sequence. For parameters, this value is the default value if you do not pass an argument value explicitly. If the data type is a single-valued data type, such as number or Boolean, the value appears in the Value column of the list view.

Single Values

You can modify the value of a single-valued data type by selecting Properties from the context menu for the variable, parameter, or property in the list view. This launches the Type Properties dialog box. Refer to the *TestStand Help* for more information about using the Type Properties dialog box.

Object References

Object reference properties can contain references to .NET or ActiveX/COM objects. They are primarily used by the .NET Adapter and the ActiveX/COM Adapter. If the variable, parameter, or property is an object reference, you can use the Type Properties dialog box to release the reference.

You can only set the reference value from within an expression, a code module using the TestStand API, or by calling the TestStand API directly using the ActiveX/COM Adapter. TestStand stores ActiveX references as an IDispatch pointer or IUnknown pointer.

The value you assign to the object reference must be a valid object pointer. Whenever you assign a non-zero value to an object reference, TestStand adds a reference to the object for as long as the variable, parameter, or property contains that value. To release the reference to the object, assign the variable, parameter, or property a new value or the constant `Nothing`. In addition, TestStand automatically releases the reference to the object when the variable, parameter, or property loses its scope. For example, if a sequence local variable contains a reference to an object, TestStand releases the reference when the call to the sequence completes. When you release all references to a .NET object, the object is marked for garbage collection. When you release all references to an ActiveX/COM object, the object is destroyed.

If you have two references, an equality comparison performs a comparison of the objects' IUnknown pointers for ActiveX and the pointer values for .NET.

Note Do not release an object variable by assigning it a value of `0`. Instead, assign the constant `Nothing` to the variable.

Arrays

If the variable, parameter, or property is an array that contains values, you access the elements of the array in the list view by selecting View Contents from the context menu. You can use the Properties item in the context menu for each array element to modify the initial value.

Dynamic Array Sizing

TestStand allows you to resize an array during execution. In an expression, you can 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, you must use the GetArrayBounds and SetArrayBounds expression functions. You can find the documentation for these functions on the Operators tab on the Expression Browser dialog box. Refer to the *TestStand Help* for more information about expressions.

In a code module, use the GetDimensions and SetDimensions methods of the PropertyObject class to obtain or set the upper and lower bounds of an array or to change the number of dimensions. Refer to the *TestStand Help* for more information about the GetDimensions and SetDimensions methods.

Using the Standard Named Data Types

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

Figure 12-5 shows the Standard Data Types tab in the Type Palette window.

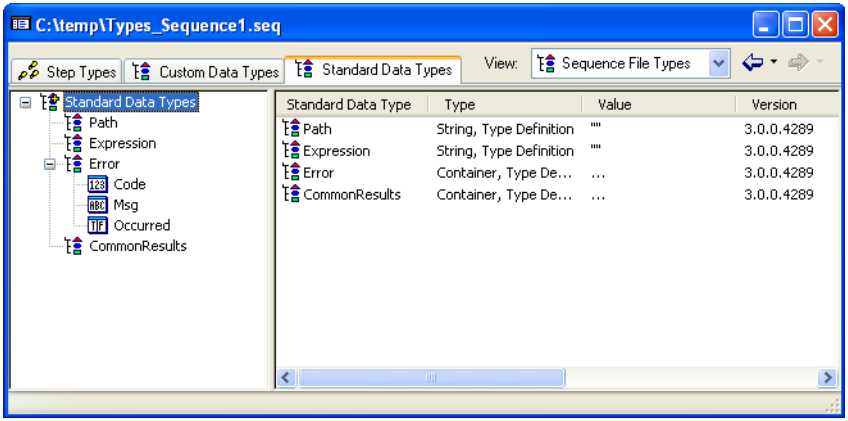


Figure 12-5. Standard Data Types Tab in the Type Palette Window

The following sections describe some of the more generally applicable standard data types.

Path

Use the Path standard data type to store a pathname as a string.

The variables, parameters, and properties that you define using the Path data type appear in the Edit Paths dialog box, which you can launch by selecting **View»Paths**. Refer to the *TestStand Help* for more information about the Edit Paths dialog box.

Error and Common Results

TestStand inserts a Results property in every step you create, regardless of whether you use a built-in step type or a custom step type. The Results property has at least three subproperties: Error, Status, and CommonResults.

The Error subproperty uses the Error standard data type. Steps in TestStand use the Error subproperty to indicate run-time errors. The Error standard data type is a container that contains three subproperties: Occurred, Code, and Msg. When a run-time error occurs in a step, the step sets the Occurred subproperty to `True`, the Code subproperty to a value that indicates that source of the error, and the Msg subproperty to a string that describes the error. You can add more subproperties to the Error standard data type. In this way, your steps can record extra run-time error information in a standard way.

The CommonResults standard data type is an object that is initially empty. By adding subproperties to it, you can add extra result information to all steps in a standard way.

If you choose to add more subproperties to Error or CommonResults, newer versions of TestStand will not overwrite them.

Be aware that if you modify CommonResults without incrementing the type version number, you may see a type conflict when you open other sequence files. These conflicts can include `FrontEndCallbacks.seq` when you are logging in or out. TestStand will automatically prompt you to increment the version number when saving changes to any data type or step type.

Creating and Modifying Custom Data Types

You can create and modify data types in the Sequence File Types view of a Sequence File window, the Global Types view of the Station Globals window, the Type Palette window, and the Types view of the User Manager window. Use the **Custom Data Types** tab to create and modify custom data types. Use the **Standard Data Types** tab to add subproperties to the standard data types.

Note The remainder of this section discusses creating and modifying custom data types on the Custom Data Types tab. The same information applies to the Standard Data Types tab.

Creating a New Custom Data Type

To create a new custom data type, select the root node in the tree view so that the existing custom data types appear in the list view. Right-click the background of the list view and select **Insert Custom Data Type** from the context menu. Figure 12-6 shows the Insert Custom Data Type submenu.

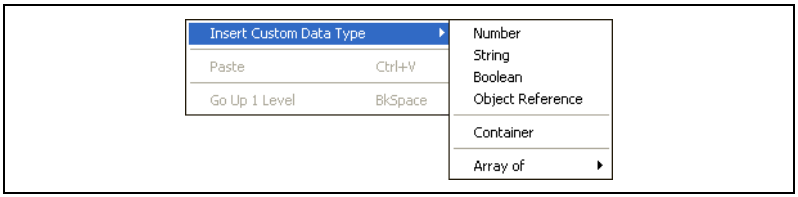


Figure 12-6. Insert Custom Data Type Submenu

The submenu gives you a set of data types from which you can select any of the simple data types that TestStand defines, including an array of any type, a container, or a custom or standard named data type.

Selecting an array type from the submenu launches the Array Bounds dialog box. Use this dialog box to specify the array bounds that TestStand applies initially to each variable, parameter, or property that you create with the data type. After you create the variable, parameter, or property, you can change its array bounds on the Bounds tab on the Type Properties dialog box. Refer to the *Modifying Data Types and Values* section of this chapter for more information about dynamically setting the size of an array.

If you select the Container type from the submenu, TestStand creates the data type without any fields.

Note When you create new data types, begin your types with a unique ID, such as a company prefix. Using a unique ID helps to prevent name collisions. For example, `NI_InstrumentConfigurationOptions` uses `NI` as a unique ID.

Customizing Built-In Data Types

You cannot modify NI-installed data types. To create a customized version of an NI-installed type, copy and rename the type in the sequence editor.

Properties Common to All Data Types

TestStand defines many properties that are common to all data types. These are called built-in data type properties. To examine and modify the values of the built-in data type properties, select **Properties** from the context menu for a data type in the List view. This launches the Data Type Properties dialog box. The Data Type Properties dialog box contains the following tabs: General, Version, Bounds, Cluster Passing, C/C++ Struct Passing, and .NET Struct Passing.

The following sections provide an overview of each tab. Refer to the *TestStand Help* for detailed information about the Data Type Properties dialog box.

General Tab

Use the **General** tab to specify an initial value and comment for the data type.

Property Flags

TestStand includes a set of property flags that you can modify. Access the Edit Flags dialog box by clicking **Advanced** in the **General** tab on the Data Type Properties dialog box. For more information about the Edit Flags dialog box, refer to the *TestStand Help*.

For a description of each of the property flag constants in the TestStand API, refer to the *PropertyFlags Constants* and the *PropertyObjTypeFlags Constants* topics in the *TestStand Help*.

Bounds Tab

Use the **Bounds** tab to define the bounds for array data types. This tab is only visible for array data types.

Version Tab

Use the **Version** tab to edit the version information for the data type.

Cluster Passing Tab

Use the **Cluster Passing** tab to define how TestStand passes instances of the data type as a cluster to LabVIEW VIs.

C/C++ Struct Passing Tab

Use the **C/C++ Struct Passing Tab** to define how TestStand passes instances of the data type as a structure to functions and methods in C/C++ DLLs.

.NET Struct Passing Tab

Use the **.NET Struct Passing** tab to define how TestStand passes instances of the data type as a structure to methods and properties in .NET assemblies.

Custom Properties of Data Types

You can add any number of fields to a data type or data type subproperty that you create as a container. To add fields to a container property in a new or existing data type, right-click the icon for the data type or a data type subproperty in the list view and select **View Contents** from the context menu. For a new data type, the list view is empty. For an existing data type, the list view displays the fields currently in the data type. Right-click the background of the list view and select **Insert Field** from the context menu. Figure 12-7 shows the Insert Field submenu.

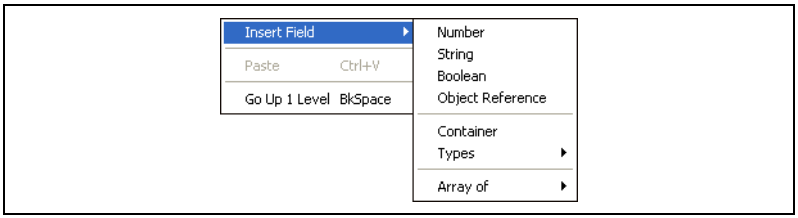


Figure 12-7. Insert Field Submenu

The submenu gives you a set of data types from which you can select any of the simple data types that TestStand defines, including an array of any type, a container, or a custom or standard named data type.

To cut, copy, paste, or rename fields, use the context menu that becomes visible when you right-click the icon for the field in the list view.

13

Creating Custom Step Types

This chapter describes how to create custom step types. For more information about types, refer to Chapter 11, *Type Concepts*. For more information about the built-in step types included in TestStand, refer to Chapter 4, *Built-In Step Types*.

Creating and Modifying Custom Step Types

TestStand gives you the flexibility to create a custom step type to fit the specific needs of your application. You can do this by modifying an existing TestStand built-in step type or creating a new step type.

If you want to change or enhance a TestStand built-in step type, copy the files to a `User` subdirectory, then copy and rename the built-in step type and its supporting modules, and make the changes to the new type and its files. This practice ensures that a newer installation of TestStand does not overwrite your customizations. It also makes it easier for you to distribute your customizations to other users.

To insert a new step type in the Type Palette window or the Sequence Files Types view, right-click the background of the list view and select **Insert Step Type** from the context menu. Use the **Copy** and **Paste** items from the context menu to copy an existing step type.

Note When you create new step types, begin your types with a unique ID, such as a company prefix. Using a unique ID will prevent name collision. For example, `NI_PropertyLoader` uses `NI` as a unique ID.

The Step Types tab on the Type Palette window shows all the step types in the selected type palette file. The Step Types tab on the Sequence File Types view of the Sequence File window only shows the step types that the steps in the sequence file use.

Figure 13-1 shows the Step Types tab in the Type Palette window.

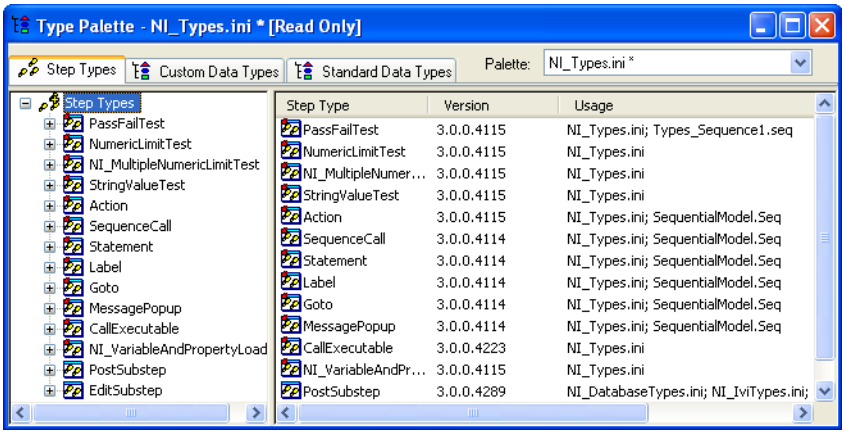


Figure 13-1. Step Types Tab in the Type Palette Window

Creating a New Custom Step Type

Complete the following steps when you are creating new step types in the Type palette:

1. Specify the menu item name for a step type on the **Menu** tab on the Step Type Properties dialog box.
2. Specify the default name for new steps you create from your type and the description expression for those steps in the **General** tab on the Step Type Properties dialog box.
3. Specify the menu item name (and button name) that invoke the editing dialog box you (optionally) define for your step type in the Edit Step section of the **Substeps** tab on the Step Type Properties dialog box.

After you install TestStand, the Step Types tab displays all the built-in step types, the Custom Data Types tab is empty, and the Standard Data Types tab contains several standard data types.

Customizing Built-In Step Types

Source code is available for the code modules that the built-in step types use as substeps. The source code project files are located in the `<TestStand>\Components\NI\StepTypes` subdirectory. If you use these source files as a starting point for step types you create, make your own copies of these files in the `<TestStand>\Components\User\StepTypes` subdirectory and rename them.

Note You cannot modify NI-installed types. To create a customized version of an NI-installed type, copy and rename the type in the sequence editor. You must also copy and rename any supporting modules from the `<TestStand>\Components\NI\StepTypes` directory to the `<TestStand>\Components\User\StepTypes` directory. Make any changes to the copy to ensure that newer installations of TestStand do not overwrite your customizations.

Properties Common to All Step Types

TestStand defines many properties that are common to all step types. These are called the built-in step type properties. Some built-in step type properties only exist in the step type itself. These are called *class step type properties*. TestStand uses the class step type properties to define how the step type works for all step instances. Step instances do not contain their own copies of the class step type properties.

Other built-in step type properties exist in each step instance. These are called *instance step type properties*. Each step you create with the step type has its own copy of the instance step type properties. TestStand uses the value you specify for an instance step type property in the step type as the initial value of the property in each new step you create.

Normally, after you create a step, you can change the values of the step's instance step type properties. However, 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 that case, you do not want the user to explicitly change the Status Expression value. TestStand uses this capability in some of the built-in step types, such as the Numeric Limit Test and String Value Test. To examine and modify the values of the built-in properties, select **Properties** from the context menu for a step type in the list view.

The Default Run Options, Default Post Actions, Default Loop Options, Default Expressions, Default Switching, and Default Synchronization tabs display instance properties. These tabs have the same appearance and behavior as the Run Options, Post Actions, Loop Options, Expressions, and Synchronization tabs of the Step Properties dialog box for a step instance. Refer to the *TestStand Help* for more information about these tabs.

Most of the properties in the other tabs are class properties. The following sections discuss each of these tabs in detail.

General Tab

Use the **General** tab to specify a name, description, and comment for the step type. You can also specify an icon and a module adapter.

Property Flags

TestStand includes a set of property flags that you can modify. Access the Edit Flags dialog box by clicking **Advanced** on the **General** tab on the Step Type Properties dialog box. Typically, you only need to configure property flags when you develop a relatively sophisticated custom step type. For more information about the Edit Flags dialog box, refer to the *TestStand Help*.

For a description of each of the property flag constants in the TestStand API, refer to the *PropertyFlags* and the *PropertyObjTypeFlags* topics of the *TestStand Help*.

Menu Tab

Use the **Menu** tab to specify the menu item name that appears for the step type in the Insert Step submenu. The Insert Step submenu appears in the context menu of individual sequence views in the Sequence File window. Use the Step Type Menu Editor to configure the organization of the Insert Step submenu. Refer to the *TestStand Help* for a description of the Step Type Menu Editor.

Substeps Tab

Use the **Substeps** tab to specify substeps for the step type. You use substeps to define standard actions, other than calling the code module, that TestStand performs for all instances of the step type. You implement a substep through a call to a code module. The code modules you call from substeps are called *substep modules*. The substeps for a step type define the editing and run-time behavior for all step instances of that type. 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 step type when configuring a particular 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. You can specify four categories of substeps for a step type:

- *Pre-Step* substeps
- *Post-Step* substeps
- *Edit* substeps
- *Custom* substeps

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

TestStand calls the Post-Step substep after calling the code module. You can specify an adapter and a module to invoke in the Post-Step substep in the Step Properties dialog box for the step type. A Post-Step substep might call a code module that compares the values the code module stored in step properties against limit values that the Edit substep stored in other step properties.

Invoke the Edit substep by selecting a menu item in the context menu for the step or by clicking the **Add** button in the Step Properties dialog box for the step. The step type specifies the name of the menu item and the caption of the button. The Edit substep launches a dialog box in which you can edit the values of the custom step properties. For example, an Edit substep might display 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 displayed by the specified Edit substep code must be modal. For example, all dialog boxes except MFC dialog boxes use the Engine.NotifyStartOfModalDialogEx and Engine.NotifyEndOfModalDialogEx methods of the TestStand API. Refer to the modal examples in the `<TestStand>\Examples\ModalDialogs` directory.

TestStand does not call Custom substeps. You can use the TestStand API to invoke a Custom substep from a code module or an operator interface.

Source code is available for many of the substep modules that the built-in step types use. You can find the source code project files in the `<TestStand>\Components\NI\StepTypes` directory. If you want to use existing step type source code as a starting point for your own step type, copy the files into the `<TestStand>\Components\User\StepTypes` directory and use unique filenames to rename the copies.

Note Threads within TestStand executions can be initialized to use either the single-threaded apartment model or the multi-threaded apartment model. TestStand executes Edit substeps in threads initialized using the single-threaded apartment model to allow the substep to open windows that contain ActiveX controls.

Disable Properties Tab

Use the **Disable Properties** tab to prevent users from modifying the settings of instance step type properties in individual steps. In this way, you make the default settings you specify in the Step Type Properties dialog box non-editable for all step instances.

The Disable Properties tab contains a list of options, in which each option represents one built-in instance property or a group of built-in instance properties. When you enable an option, you prevent users from modifying the value of the corresponding property or group of properties.

Code Templates Tab

Use the **Code Templates** tab to associate one or more code templates with the step type. A code template is a set of source files that contain skeleton code. The skeleton code serves as a starting point for the development of code modules for steps that use the step type. TestStand uses the code template when you click **Create Code** in the Specify Module dialog box for a step.

TestStand comes with default code templates that you can use for any step type. You can customize code templates for individual step types. For the Numeric Limit Test step type, for instance, you might want to include example code for accessing the high- and low-limit properties in a step.

Template Files for Different Development Environments

Because different module adapters require different types of code modules, code templates typically correspond to a particular programming language in a specific development environment. In versions of TestStand prior to TestStand 3.0, code templates that were designed for use with the LabVIEW and LabWindows/CVI Adapters contained multiple source files

for use in those environments. For example, a previous default code template included one `.c` file for the LabWindows/CVI Adapter and eight VIs for the LabVIEW Adapter, where the multiple VIs corresponded to the different combinations of parameter options you can set in the Edit LabVIEW VI Call dialog box.

TestStand 3.0 refers to these code templates as *legacy* code templates. They are included to provide backwards compatibility with previous versions of TestStand. TestStand 3.0 also includes new code templates that support the new features of the LabVIEW and LabWindows/CVI Adapters.

TestStand uses the name of a directory within the `<TestStand>\CodeTemplates\NI` or `<TestStand>\CodeTemplates\User` directory as the code template name. TestStand stores the source file for the module adapter in the directory. TestStand also stores a `.ini` file in each subdirectory that contains parameter information and a description string that TestStand displays for the code template. Table 13-1 lists the subdirectories that contain the default code templates for each development environment.

Table 13-1. Locations of Default Code Templates

Subdirectory Name	Template Description
`DefaultTemplate`	Legacy default template
`DefaultC++.NET`	Default template for C++ in Microsoft Visual Studio .NET
`DefaultCSharp.NET`	Default template for C# in Microsoft Visual Studio .NET
`DefaultCVI`	Default template for C in LabWindows/CVI
`DefaultHTB72_Template`	Default template for HTBasic 7.2
`DefaultHTB80_Template`	Default template for HTBasic 8.0
`DefaultLabVIEW`	Default template for LabVIEW 7.0 or later
`DefaultVB.NET`	Default template for Microsoft Visual Basic .NET
`DefaultVC++_Template`	Default template for C++ in Microsoft Visual Studio 6.0

Code templates for the LabVIEW, LabWindows/CVI, and C/C++ DLL Adapters can have any number of parameters that are compatible with the data types you can specify in the Specify Module dialog box for those adapters.

Legacy code templates for the LabVIEW Adapter always specify **Test Data** and **error out** clusters as parameters. The eight different VIs for each LabVIEW Adapter legacy code template specify various combinations of the Input Buffer, Invocation Information, and SequenceContext parameters. When TestStand uses a legacy LabVIEW template VI to create skeleton code, it chooses the correct VI to use according to the current settings in the Optional Parameters dialog box.

Legacy code templates for the LabWindows/CVI Adapter always specify two parameters: a pointer to a **tTestData** structure and a pointer to a **tTestError** structure. When TestStand uses a legacy LabWindows/CVI template module to create skeleton code, it validates the function prototype in the template module against this requirement. TestStand reports an error if the prototype is incorrect.

When TestStand uses a code template for a DLL to create skeleton code, it compares the parameter list in the source file to the parameter information on the Module tab. If these sets of information do not match, TestStand prompts you to select which prototype to use for the skeleton code. If you choose to use the prototype from the template source file, you can also request that TestStand update the Module tab to match the 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.

You can specify entries for TestStand to place in the Value controls, as described in the *TestStand Help*. TestStand stores this information in the `.ini` file in the template subdirectory.

Creating and Customizing Code Template Files

Use the **Code Templates** tab to create a new code template. TestStand prompts you to specify a subdirectory name and an existing code template as a starting point. TestStand copies the files for the existing code template into the new subdirectory in the `<TestStand>\CodeTemplates\User` directory and changes the names. Then, you must modify the code template files in order to customize them.

You can customize the code template files to include example code that helps the test developer learn how to access the important custom properties of the step. For most environments, you can add a value parameter to pass the information from TestStand. For example, to show how to obtain the high- and low-limit properties in a LabVIEW or LabWindows/CVI code template for a Numeric Limit Test step, you may customize the prototype for the code module by specifying the high and low limits as value parameters.

As another example, you might want to show how to return a measurement value from a code module. For the LabVIEW, LabWindows/CVI, and C/C++ DLL Adapters, you can customize the prototype in the code template by specifying the measurement as a reference parameter.

Multiple Code Templates per Step Type

You can specify more than one code template for a step type. For example, you might want to have code templates that contain example code for conducting the same type of tests with different types of instruments or data acquisition boards. When a step type has multiple code templates and you click **Create Code** in the Specify Module dialog box, TestStand prompts you to choose from a list of templates.

Version Tab

The Version tab for a Step Type Properties dialog box is identical to the Version tab you use on the Type Properties dialog box for a custom data type. Refer to the *TestStand Help* for more information about the Version tab.

Custom Properties of Step Types

You can define any number of custom properties in a step type so that each step you create with that step type uses those custom properties.

Open the nodes in the tree view of the Step Types tab to display all step types and their custom properties for the selected file. To display the custom properties of a step type and its subproperties in the list view, select the node for the step type and the custom property, respectively, in the tree view.

From the list view, select **View Contents** from the context menu to display the contents of a step type or property. To display the contents of the next highest level, press **<Backspace>** in either the tree view or the list view, or select **Go Up 1 Level** from the context menu in the list view background.

Figure 13-2 shows the custom properties for the Numeric Limits step.

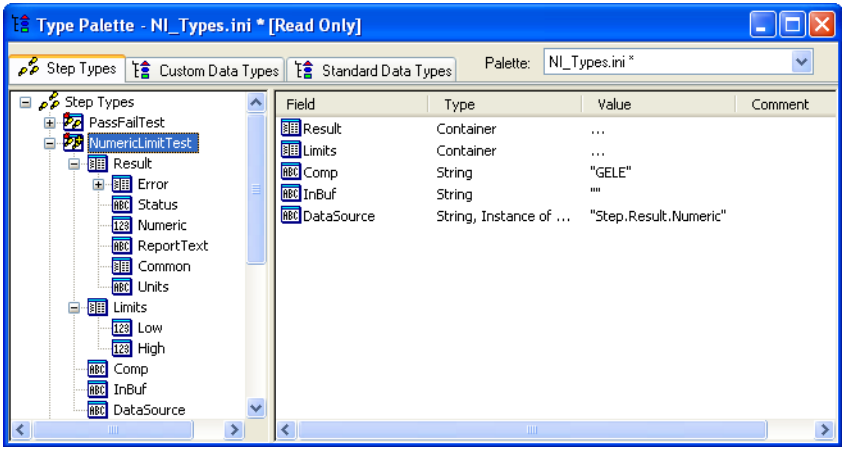


Figure 13-2. Custom Properties of a Step Type

14

Deploying TestStand Systems

This chapter describes the TestStand Deployment Utility and the steps necessary to successfully deploy a TestStand system from a development computer to one or more target computers.

TestStand System Components

TestStand systems are composed of a variety of components that work together to create the entire system. These components can include the following:

- TestStand Engine and its supporting files
- LabVIEW and LabWindows/CVI Run-Time Engines
- Process models and their supporting files
- Step types and their supporting files
- Configuration files
- Operator interface applications
- Workspace files
- Sequence files
- Code modules and their supporting files
- Hardware drivers

When deploying a TestStand system from a development computer to a target computer, you must ensure that all of the components that your system uses are deployed to the target computer. TestStand provides the TestStand Deployment Utility to assist you with this process.

TestStand Deployment Utility

The TestStand Deployment Utility simplifies the complex process of deploying a TestStand system by automating many of the steps involved in deployment, including collecting sequence files, code modules, and support files for your test system and then creating an installer for these files.

Setting Up the TestStand Deployment Utility

Complete the following steps to deploy a TestStand test system using the TestStand Deployment Utility:

1. Identify the components to deploy.
2. Determine whether to create an installer for your system.
3. Create a system workspace file, if necessary.
4. Configure and build the deployment.

The following sections discuss each of these steps.

Identifying Components for Deployment

The TestStand Deployment Utility can create installable images, which are directories of files to be installed to the target computer, of the following TestStand components:

- Components located in the `<TestStand>\...\User` subdirectories.
- A TestStand workspace file and its dependent files, including sequence files, code modules, and so on.

Additionally, the deployment utility can create an installer that installs these components with the TestStand Engine, plus components in the `<TestStand>\...\NI` subdirectories.

Determining Whether to Create an Installer With the TestStand Deployment Utility

If you plan to deploy the TestStand Engine and the TestStand components located in the `<TestStand>\...\NI` subdirectories, you must use the TestStand Deployment Utility to create an installer.

You do not need to use the deployment utility to create an installer if you plan to deploy your TestStand test system using a third party installer development tool, such as Wise or InstallShield, or by using a source code or revision control system to deploy your system files to target computers.

Creating a System Workspace File

Before deploying your sequence files and code modules, you must create a workspace file that contains all of the sequence files that your test system could execute. The deployment utility analyzes those sequence files to determine which files they reference, such as code module files. Also, add any files that are not stored in a `<TestStand>\...\User` subdirectory or files that are not referenced by your sequence files to your workspace file, such as the support files required by code module DLLs.

If you are using the TestStand Deployment Utility to deploy only the TestStand Engine or the components located in the `<TestStand>\...\User` subdirectories, you do not need to create a workspace file for your test system.

Refer to Chapter 2, *Sequence Files and Workspaces*, for more information about TestStand workspace files.

Configuring and Building the Deployment

Within the sequence editor, select **Tools»Deploy TestStand System** to launch the TestStand Deployment Utility. This launches the Deploy TestStand System dialog box, in which you can configure the settings for deploying your test system, including the components to install and installer settings.

Refer to the *TestStand Help* for more information about the Deploy TestStand System dialog box.

Using the TestStand Deployment Utility

This section describes how the TestStand Deployment Utility builds a deployable test system.

File Collection

When deploying a workspace file, the deployment utility analyzes the workspace for any dependent files. For example, if your workspace contains a sequence file, the deployment utility searches the steps in every sequence of the file to find the referenced code modules. This analysis continues recursively until all files in the workspace hierarchy are analyzed.

Since automatically distributing every file used by your sequences could be problematic, the deployment utility includes a filtering function that removes potentially unwanted files. For example, if you have steps in your sequences that call functions in Windows system DLLs, the deployment utility will not deploy those DLLs to the target computer.

The `Filter.ini` file, located in the `<TestStand>\Components\NI\Tools\Deployment Utility` directory, defines those files that the deployment utility automatically excludes from any deployment package it creates. By default, the deployment utility does not deploy any files located in the `<TestStand>\Bin` or `<TestStand>\...\NI` directories. Additionally, it does not deploy any `.exe` or `.dll` files located in the `<Windows>` or `<Windows>\System32` directories.

You may add automatically excluded files to your workspace file, but do so with caution to prevent incompatibility issues. For example, if your development computer operates on Windows XP, and you deploy a Windows system DLL from that computer to a target computer running Windows 2000, you will likely experience DLL version incompatibility issues.

Note The TestStand Deployment Utility does not deploy .NET or ActiveX/COM code modules. You must manually add these code modules and their supporting files to the system workspace or install them separately on the target computer.

VI Processing

The deployment utility analyzes all of the LabVIEW VIs that it deploys to determine their complete hierarchies, including all subVIs, DLLs, external subroutines, run-time menus, LabVIEW Express configuration diagrams, and help files that your VIs may reference. It then packages these VIs and their hierarchies to ensure that they will be executable on systems that do not have the LabVIEW development system installed.

Note You must have the LabVIEW development system installed on your development computer in order for the TestStand Deployment Utility to perform VI processing.

Note If your VIs call other VIs dynamically using VI Server, you must add those VIs manually to your system workspace file.

Sequence File Processing

The TestStand Deployment Utility also performs processing on sequence files in order to remove absolute paths. Absolute paths that are functional on your development computer may be invalid on the target computer, especially if the base installation directories are different.

For example, if you have installed TestStand to `c:\TestStand` on your development system and to `c:\Program Files\National Instruments\TestStand` on your target computer, the absolute path `c:\TestStand\test.dll` will be valid on your development computer but invalid on your target computer.

The deployment utility corrects this potential problem by changing absolute path references in sequence files to relative paths that initiate from one of the following search directories:

- Current sequence file directory
- TestStand installation directory
- `Windows\System32` directory
- `Windows` directory

If the target file is located outside of these directories, TestStand uses a path that is relative to the installation directory and then adds the installation directory to the list of default search paths during the installation.

Guidelines for Successful Deployment

Follow these guidelines to ensure that your deployment process is successful:

- **Use unique file names**—Always use unique file names, even if you are working with a revision of an existing file. Ambiguous file names can cause the deployment utility to locate incorrect files, which can result in incorrect behavior.
- **Use relative paths and search paths**—Relative paths allow TestStand to find files even if they were installed in a different location on the target computer than they were on the development computer. For example, you can locate a file that was saved in `c:\Program Files\National Instruments\TestStand 3.0\Reports` on the development computer and in `c:\TestStand\Reports` on the target computer using the relative path `Reports`, since the `TestStand` installation directory is included in the default search path.

- **Manually add any additional search paths to the list of default search paths on the target computer**—You must manually add search paths to the list of default search paths. The TestStand Deployment Utility will not copy additional search paths since the new directories may not exist on the target computer. Also, ambiguous file names in these search paths can cause TestStand to locate the wrong file.
- **Manually add dynamically-referenced files to your workspace**—Dynamically-referenced files include any sequences specified by an expression, property loader files specified by expressions, LabVIEW VIs called using VI Server, and dynamically-loaded DLLs.
- **Manually add any supporting DLLs required by your code modules to your workspace**—Do not add any DLLs that are part of TestStand or your operating system.

Common Deployment Scenarios

The following examples describe how to use the TestStand Deployment Utility in common deployment scenarios. To complete the examples, you will need one development computer containing a complete installation of TestStand and one target computer.

Deploying the TestStand Engine

1. Launch the TestStand Deployment Utility by selecting **Tools»Deploy TestStand System** from within the sequence editor.
2. On the **System Source** tab, enable the **Deploy Files in TestStand User Directories** option.

   This option collects files from the `<TestStand>\...\User` directories, so that any customizations that you have made to process models, step types, language strings, and so on, will be distributed to the target computer.
3. On the **Installer Options** tab, enable the **Install TestStand Engine** option.
4. On the **Installer Options** tab, click **Engine Options** to launch the TestStand Engine Options dialog box, which you use to select the TestStand components that should be present in the installer.

5. In the TestStand Engine Options dialog box, expand **Operator Interfaces»Full-Featured** in the tree view.
   a. Click the **X** next to `LabWindows/CVI` to include the Full-Featured LabWindows/CVI Operator Interface in the engine installation. The X should become a green checkmark.
   b. Click **OK** to accept the new settings and close the dialog box.
6. Click **Save** and save this build as `EngineInstaller.tsd`.
7. Click **Build** to create the installer.
8. To use the installer, copy all of the files from the `<My Documents>\TestStand Deployment\Installer` directory to a CD or to a shared directory on your network.
9. Go to your target computer and insert the CD or connect to the network, and then run the `setup.exe` application to start the installer.
10. Select the target directory in which to install the TestStand Engine. Click **Next** to begin the installation.
11. Once the installation is complete, run the LabWindows/CVI Operator Interface by selecting **Start»Programs»National Instruments»TestStand 3.0»Operator Interfaces»LabWindows/CVI**. Verify that the TestStand Engine was installed correctly.

Distributing Tests From a Workspace

1. Launch the TestStand Deployment Utility by selecting **Tools»Deploy TestStand System** from within the sequence editor.
2. On the **System Source** tab, enable the **Deploy Files from TestStand Workspace File** option.
3. Click the **File Browse** button, which is located next to the Workspace File Path control.
4. Browse to the `<TestStand>\Examples\Deployment` directory and select the `Test.tsw` workspace file. Click **Open**.
5. Select the **Distributed Files** tab. A dialog box launches to request permission to analyze the source files. Click **Yes**.

   The deployment utility analyzes the workspace file and its dependent files.
6. Locate `Unused.dll` in the tree view. This DLL is not used by the test system. Click the checkmark located next to the file in the tree view to remove it from the distribution.
7. On the **Installer Options** tab, enable the **Install TestStand Engine** option.

8. In the TestStand Engine Options dialog box, expand **Operator Interfaces»Full-Featured** in the tree view.
   a. Click the **X** next to `LabVIEW` to include the Full-Featured LabVIEW Operator Interface in the engine installation. The X should become a green checkmark.
   b. Click **OK** to accept the new settings and close the dialog box.
9. Click **Save** to save this build as `Test.tsd`.
10. Click **Build** to create an installer.
11. To use the installer, copy all of the files from the `<My Documents>\TestStand\Deployment\Installer` directory to a CD or to a shared directory on your network.
12. Go to your target computer and insert the CD or connect to the network, and then run the `setup.exe` application to start the installer.
13. Select a target directory in which to install the tests, and click **Next** to begin the installation.
14. Once the installation is complete, launch the LabVIEW Operator Interface by selecting **Start»Programs»National Instruments» TestStand 3.0»Operator Interfaces»LabVIEW**.
15. Load and run `<TestStand>\Examples\Deployment\test.seq` to verify the installation.

Adding Dynamically Called Files to a Workspace

1. Start the TestStand Deployment Utility by selecting **Tools»Deploy TestStand System** from within the sequence editor.
2. On the **System Source** tab, enable the **Deploy Files from TestStand Workspace File** option.
3. Click the **File Browse** button, which is located next to the Workspace File Path control.
4. Browse to the `<TestStand>\Examples\Deployment` directory and select the `Dynamically_called_sequence.tsw` workspace file. Click **Open**.
5. Select the **Distributed Files** tab. A dialog box launches to request permission to analyze the source files. Click **Yes**.

   The deployment utility analyzes the workspace file and its dependent files.

Note You should receive a warning in the Status Log on the Build Status tab stating that the sequence was called using an expression.

The deployment utility processes the workspace and updates the deployed files list. Notice that `dynamic.seq` is not included in the list.

6. From within the sequence editor, load the following workspace file: `<TestStand>\Examples\Deployment\Dynamically_called_sequence.tsw`.
7. Add `<TestStand>\Examples\Deployment\dynamic.seq` to this workspace file and then save the changes to the workspace.
8. On the **Distributed Files** tab on the TestStand Deployment Utility, click **Analyze Source Files** to analyze the modified workspace file.

   The deployment utility analyzes the workspace file and its dependent files.

Note You will receive another warning in the Status Log of the Build Status tab stating that the sequence was called using an expression. You can ignore this second warning, since you have just added the sequence to the workspace.

9. Notice that `dynamic.seq` is now included in the distributed file list.
10. On the **Installer Options** tab, enable the **Install TestStand Engine** option.
11. In the TestStand Engine Options dialog box, expand **Operator Interfaces»Full-Featured** in the tree view.
    a. Click the **X** next to `C++ (MFC)` to include the Full-Featured C++ (MFC) Operator Interface in the engine installation. The X should become a green checkmark.
    b. Click **OK** to accept the new settings and close the dialog box.
12. Click **Save** to save the build as `Dynamic.tsd`.
13. Click **Build** to create an installer.
14. To use the installer, copy all of the files from the `<My Documents>\TestStand\Deployment\Installer` directory to a CD or to a shared directory on your network.
15. Go to your target computer and insert the CD or connect to the network, and then run the `setup.exe` application to start the installer.
16. Select the target directory where you want to install the tests, and click **Next** to begin the installation.

17. Once the installation is complete, launch the C++ (MFC) Operator Interface by selecting **Start»Programs»National Instruments» TestStand 3.0»Operator Interfaces»C++ (MFC)**.
18. Load and run `<TestStand>\Examples\Deployment\Call_sequence_dynamically.seq` to verify the installation.

Distributing an Operator Interface

Before completing this exercise, copy the files in `<TestStand>\OperatorInterfaces\NI\Simple\CVI` to `<TestStand>\OperatorInterfaces\User\Simple\CVI`.

1. Within the sequence editor, select **File»New Workspace** to create a new workspace file. Save this workspace as `Deploy Operator Interface.tsw`
2. In the Workspace window, right-click the Workspace icon and select **Insert New Project into Workspace** from the context menu. Save this project as `Operator Interface.tpj`.
3. In the Workspace window, right-click the Project icon and select **Add Files to Project** from the context menu.
4. In the file browse dialog box, browse to `<TestStand>\OperatorInterfaces\User\Simple\CVI\` and change the **Files of Type** setting to **All Files (*.*)**.
5. Select both `TestStand.exe` and `TestExec.uir` and click **Add**.

Note If you are prompted to resolve the path, select **Use a relative path for the file you selected**. Enable the **Apply to All** option, and then click **OK** twice to close the open dialog boxes.

6. Select **File»Save** to save the workspace file.
7. Start the TestStand Deployment Utility by selecting **Tools»Deploy TestStand System** from within the sequence editor.
8. On the **System Source** tab, enable the **Deploy Files From TestStand Workspace File** option.
9. Browse to the workspace file you saved in Step 6. Click **Open**.
10. Select the **Distributed Files** tab and click **Yes** in the dialog box requesting permission to analyze the workspace files.

    The deployment utility analyzes the workspace file and its dependent files.

11. Locate `TestExec.exe` in the tree view and click on the file. The File Properties section to the right of the tree view should update to reflect this selection.
12. Enable the **Create Program Item** option and type `Simple CVI` into the neighboring string field to add a shortcut menu item for `TestExec.exe`.
13. On the **Installer Options** tab, enable the **Install TestStand Engine** option.
14. Click **Save** to save the build as `SimpleCVIOI.tsd`.
15. Click **Build** to create an installer.
16. To use the installer, copy all of the files from the `<My Documents>\TestStand\Deployment\Installer` directory to a CD or to a shared directory on your network.
17. Go to your target computer and insert the CD or connect to the network, and then run the `setup.exe` application to start the installer.
18. Select the target directory where you want to install the tests, and click **Next** to begin the installation.
19. Once the installation is complete, load and run the Simple CVI operator interface from the **Start»Programs»My TestStand System» Simple CVI** program group to verify the installation.

You have completed this example. For more information about the TestStand Deployment Utility and using the Deploy TestStand System dialog box, refer to the *TestStand Help*.

A

Process Model Architecture

This appendix discusses the purpose and usage of the process models that come with TestStand. It also describes the directory structure that TestStand uses for process model files and the special capabilities that the TestStand Sequence Editor has for editing process model sequence files.

To better understand the information in this appendix, review the *Process Models* section of Chapter 1, *TestStand Architecture*, which discusses the purpose of process models, entry points, and the relationship between a process model and a client sequence file.

TestStand Process Model Architecture

The Sequential, Parallel, and Batch process models all have the same basic structure for running a test sequence. Using the Test UUTs or Single Pass entry point, the process models run test sequences, generate reports, and log UUT results to a database according to your configuration settings. Figure A-1 illustrates the basic processes that these models follow.

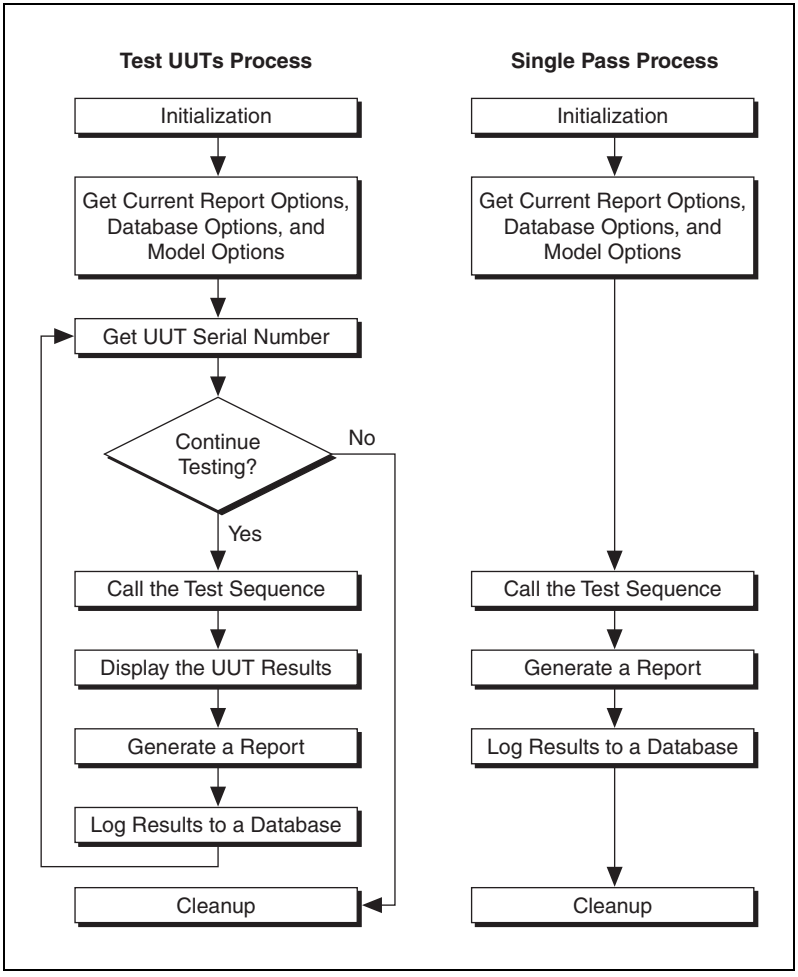


Figure A-1. Process Flow

The main differences between the process models are the number of UUTs that each process model runs for the Test UTTs or Single Pass entry points and the way each process model relates to and synchronizes with UUTs.

TestStand Process Models

Table A-1 lists the TestStand process models and their respective sequence files.

Table A-1. TestStand Process Models

Process Model	Process Model Sequence File
Sequential Model	`<TestStand>\Components\NI\Models\TestStandModels\SequentialModel.seq`
Batch Model	`<TestStand>\Components\NI\Models\TestStandModels\BatchModel.seq`
Parallel Model	`<TestStand>\Components\NI\Models\TestStandModels\ParallelModel.seq`

The Sequential model is the default TestStand process model. The Batch and Parallel models have features to help you implement test stations that test multiple UUTs at the same time.

You can create your own process models or modify a copy of a process model that TestStand provides.

Features Common to all TestStand Process Models

All TestStand process models identify UUTs, generate test reports, log results to databases, and display UUT status information. These process models also allow client sequence files to customize various model operations by overriding model-defined callback sequences.

Process models provide Configuration and Execution entry points which you can use to configure model settings and to run client files under the model. Model entry points are typically listed in an application under the Configure and Execute menus.

TestStand process models have the following Execution entry points:

- **Single Pass**—Tests one UUT or a single batch of UUTs without identifying them.
- **Test UUTs**—Tests and identifies multiple UUTs or UUT batches in a loop.

Note When you select the Test UUTs entry point to start an execution that continuously tests UUTs, any configuration changes that you make to the Report, Database, or Model Options entry points will not affect UUTs tested in that execution.

TestStand process models have the following Configuration entry points:

- **Report Options**—Launches the Report Options dialog box, in which you can configure the location and contents of report files.
- **Database Options**—Launches the Database Options dialog box, in which you can configure the logging of results to database tables.
- **Model Options**—Launches the Model Options dialog box, in which you can configure the number of test sockets and other options related to process models.

For more information about the dialog boxes associated with the Configuration entry points, refer to the *TestStand Help*.

Sequential Model

The most basic process model is the Sequential process model. The Sequential process model tests one UUT at a time.

Parallel and Batch Models

The Parallel and Batch models have features that make it easier to simultaneously test groups of similar UUTs. Use these models to run the same test sequence on multiple UUTs at the same time.

For both the Parallel and Batch models, specify the number of test sockets in your system in the Model Options dialog box, which you can access by selecting **Configure»Model Options**.

Parallel Model

Use the Parallel model to control multiple independent test sockets. The Parallel model allows you to start and stop testing on any test socket at any time. For example, if you have five test sockets for testing radios, the Parallel model allows you to load a new radio into an open test socket while the other test sockets are testing other radios.

When you select the Single Pass entry point, the Parallel model launches a separate execution for each test socket without prompting for UUT serial numbers.

Batch Model

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

When you select the Single Pass entry point, the Batch model launches a separate execution for each test socket without prompting for UUT serial numbers.

Selecting the Default Process Model

To change your default process model, select **Configure»Station Options** and click the **Model** tab. Select a model from the from the Station Model ring control or click **Browse** to select a process model sequence file. You can also use the Sequence File Properties dialog box to specify that a sequence file always uses a particular process model.

Directory Structure for Process Model Files

The TestStand installer places process model files in the <TestStand>\Components\NI\Models\TestStandModels directory.

If you want to modify a TestStand process model, copy the TestStandModels directory to a new subdirectory under the <TestStand>\Components\User\Models directory. In the new directory, rename the process model sequence files and any code module files. Next, update the process model sequence file you are customizing to call the modules with the new file names you select. By placing your modifications under <TestStand>\Components\User, you ensure that a newer installation of TestStand does not overwrite your customizations.

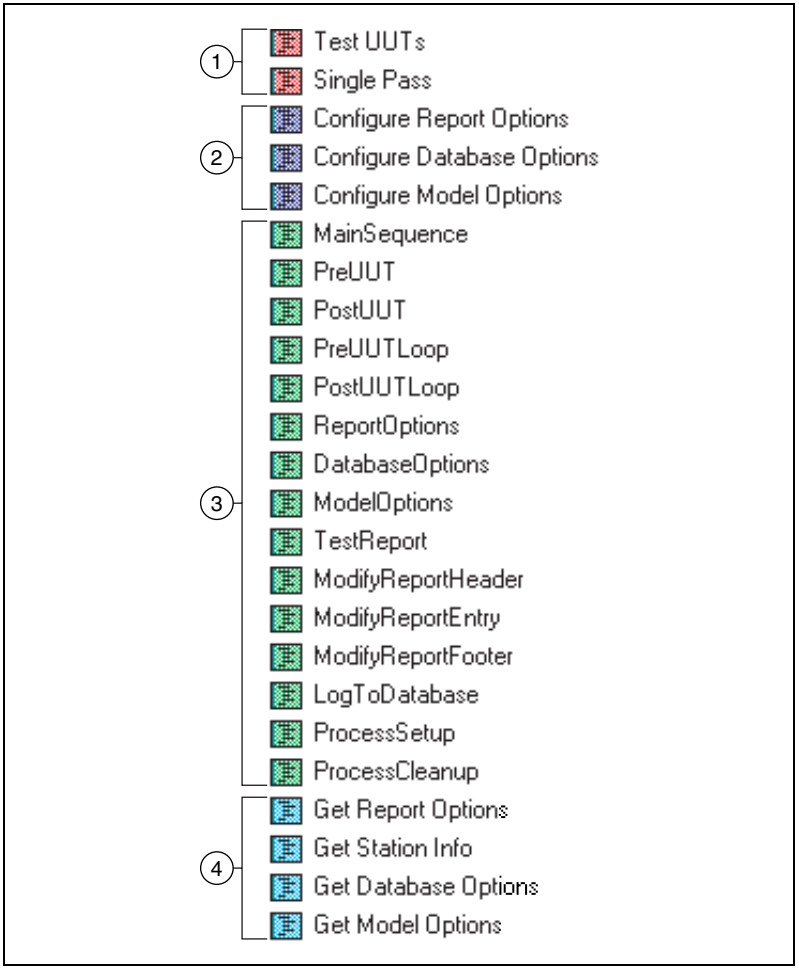
The list of search paths in TestStand includes the subdirectories in <TestStand>\Components\User. The <TestStand>\Components\User directory protects your customized components and serves as the staging area for the components that you include in your own run-time distribution of TestStand.

When you create a custom process model, you must first establish your custom process model sequence file as the process model for the station. Make this assignment on the **Model** tab on the Station Options dialog box.

Sequential Process Model

Sequences

Figure A-2 shows a list of all the sequences found in the Sequential process model, SequentialModel.seq. The sequences are divided into four categories: Execution entry points, Configuration entry points, Model callbacks, and Utility subsequences.



1	Execution Entry Points	3	Model Callbacks
2	Configuration Entry Points	4	Utility Subsequences

Figure A-2. Sequences in the Sequential Process Model

Execution Entry Points

The following sequences are Execution entry points in the Sequential process model:

- **Test UUTs**—Initiates a loop that repeatedly identifies and tests UUTs. When a window for a client sequence file is active, the Test UUTs item is listed in the Execute menu. For more information about the Test UUTs entry point, refer to *Test UUTs* in the *Sequential Process Model* section of this appendix.
- **Single Pass**—Tests a single UUT without identifying it. In essence, the Single Pass entry point performs a single iteration of the loop that the Test UUTs entry point performs. When a window for a client sequence file is active, the Single Pass item is listed in the Execute menu. For more information about the Single Pass entry point, refer to *Single Pass* in the *Sequential Process Model* section of this appendix.

Configuration Entry Points

The following sequences are Configuration entry points in the Sequential process model:

- **Configure Report Options**—Launches the Report Options dialog box, in which you can specify the contents, format, and pathname of the test report. The settings in the Report Options dialog box apply to the test station as a whole. The entry point saves the station report options to disk. The entry point item is listed as Report Options in the Configure menu. For more information about report options, refer to Chapter 6, *Database Logging and Report Generation*.
- **Configure Database Options**—Launches the Database Options dialog box, in which you can specify the database logging options. The settings in the Database Options dialog box apply to the test station as a whole. The entry point saves the station database options to disk. The entry point item is listed as Database Options in the Configure menu. For more information about database options, refer to Chapter 6, *Database Logging and Report Generation*.
- **Configure Model Options**—Launches the Model Options dialog box, in which you can specify model options other than database or report options. The settings in the Model Options dialog box apply to the test station as a whole. The entry point saves the station model options to disk. The entry point item is listed as Model Options in the Configure menu.

Model Callbacks

The following sequences are Model callbacks in the Sequential process model, which you can override in a client sequence file:

- **MainSequence**—Test UUTs and Single Pass call this callback to test a UUT. The MainSequence callback is empty in the process model file. The client sequence file must contain a MainSequence callback that performs the tests on a UUT.
- **PreUUT**—Launches the UUT Information dialog box, which the operator uses to enter the UUT serial number. The Test UUTs entry point calls the PreUUT callback at the beginning of each iteration of the UUT loop. If the operator indicates through the dialog box that no more UUTs are available for testing, the UUT loop terminates. If the operator chooses to stop testing, the IdentifyUUT step sets the ContinueTesting parameter to `False`.

  The ContinueTesting parameter is a local variable that the Test UUTs sequence passes to the PreUUT Callback sequence. If the operator enters a UUT serial number, the IdentifyUUT step stores the serial number in the UUT.SerialNumber parameter, which is a local variable that the Test UUTs sequence passes to the PreUUT Callback sequence.
- **PostUUT**—Displays a banner indicating the status of the test that the MainSequence callback in the client sequence file performs on the UUT. The Test UUTs entry point calls the PostUUT callback at the end of each iteration of the UUT loop.
- **PreUUTLoop**—The Test UUTs entry point calls this callback before the UUT loop begins. The PreUUTLoop callback in the process model file is empty.
- **PostUUTLoop**—The Test UUTs entry point calls this callback after the UUT loop terminates. The PostUUTLoop callback in the process model file is empty.
- **ReportOptions**—Execution entry points call this callback through the GetReportOptions subsequence. After reading the test station report options from disk, GetReportOptions calls the ReportOptions callback to give the client sequence file an opportunity to modify the report options. For example, you might want to force the report format to be ASCII-text for a particular client sequence file. The ReportOptions callback in the process model file is empty.
- **DatabaseOptions**—Execution entry points call this callback through the GetDatabaseOptions subsequence. After reading the test station database options from disk, GetDatabaseOptions calls the DatabaseOptions callback to give the client sequence file an

opportunity to modify the database options. The DatabaseOptions callback in the process model file is empty.

- **ModelOptions**—Execution entry points call this callback through the GetModelOptions subsequence. After reading the test station model options from disk, GetModelOptions calls the ModelOptions callback to give the client sequence file an opportunity to modify the model options. The ModelOptions callback in the process model file is empty.
- **TestReport**—Execution entry points call this callback to generate the contents of the test report for one UUT. You can override the TestReport callback in the client sequence file if you want to change its behavior entirely. The default process model defines a test report for a single UUT as a header, an entry for each step result, and a footer. If you do not override the TestReport callback, you can override the ModifyReportHeader, ModifyReportEntry, and ModifyReportFooter callbacks to customize the test report.

  Depending on the settings in the Report Options dialog box, the TestReport callback determines whether TestStand builds the report body using sequences or a DLL. If you select the Sequence option, the TestReport callback calls the AddReportBody sequence in `reportgen_xml.seq`, `reportgen_html.seq`, or `reportgen_txt.seq` to build the report body. The sequence report generator uses a series of sequences with steps that recursively process the result list for the execution. If you select the DLL option, the TestReport callback calls a single function in `modelsupport2.dll` to build the entire report body before returning. You can access the project and source code for the DLL built in LabWindows/CVI from the `<TestStand>\Components\NI\Models\TestStandModels` directory.
- **ModifyReportHeader**—The TestReport callback calls this callback in order to modify the report header using the client sequence file. ModifyReportHeader receives the following parameters: the UUT, the tentative report header text, and the report options. The ModifyReportHeader callback in the process model file is empty.
- **ModifyReportEntry**—The TestReport callback calls this callback in order to modify the entry point for each step result using the client sequence file. Using subsequences, the TestReport callback calls ModifyReportEntry for each result in the result list for the UUT. ModifyReportEntry receives the following parameters: an entry from the result list, the UUT, the tentative report entry text, the report options, and a level number that indicates the call stack depth at the time the step executed. The ModifyReportEntry callback in the process model file is empty.

Note In the Report Options dialog box, you can choose to use sequences or a DLL to produce the report body. If you select the DLL option, TestStand generates reports more efficiently. However, TestStand will not call ModifyReportEntry callbacks if the DLL option is enabled.

- **ModifyReportFooter**—The TestReport callback calls this callback in order to modify the report footer using the client sequence file. ModifyReportFooter receives the following parameters: the UUT, the tentative report footer text, and the report options. The ModifyReportFooter callback in the process model file is empty.
- **LogToDatabase**—Execution entry points call this callback to populate a database with the results for one UUT. You can override the LogToDatabase callback in the client sequence file if you want to change its behavior entirely. LogToDatabase receives the following parameters: the UUT, the result list for the UUT, and the database options.
- **Process Setup**—Execution entry points call this callback from the Setup step groups to give the client sequence file an opportunity to execute any setup steps that must run only once during the execution of the process model.
- **Process Cleanup**—Execution entry points call this callback from the Cleanup step groups to give the client sequence file an opportunity to execute any cleanup steps that must run only once during the execution of the process model.

Utility Subsequences

The following sequences are Utility subsequences that are called by the other sequences in the Sequential process model:

- **Get Report Options**—Execution entry points call this sequence at the beginning of an execution. Get Report Options reads the report options and then calls the ReportOptions callback to give you an opportunity to modify the report options in the client sequence file.
- **Get Station Info**—Execution entry points call this sequence at the beginning of an execution. Get Station Info identifies the test station name and the current user.

- **Get Database Options**—Execution entry points call this sequence at the beginning of an execution. Get Database Options reads the database options and then calls the DatabaseOptions callback to give you an opportunity to modify the database options in the client sequence file.
- **Get Model Options**—Execution entry points call this sequence at the beginning of an execution. Get Model Options reads the model options and then calls the ModelOptions callback to give you an opportunity to modify the model options in the client sequence file.

Test UUTs

Table A-2 lists the most significant steps of the Test UUTs entry point in the Sequential process model, in the order that the Test UUTs entry point performs them.

Table A-2. Order of Actions the Sequential Process Model Test UUTs Entry Point Performs

Action Number	Description	Remarks
1	Call PreUUTLoop callback.	Callback in the process model file is empty.
2	Call Get Model Options Utility subsequence.	Reads model options from disk. Calls the ModelOptions callback to give the client sequence file an opportunity to modify the model options.
3	Call Get Station Info Utility subsequence.	Identifies the test station name and the current user.
4	Call Get Report Options Utility subsequence.	Reads report options from disk. Calls the ReportOptions callback to give the client sequence file an opportunity to modify the report options.
5	Call Get Database Options Utility subsequence.	Reads database options from disk. Calls the DatabaseOptions callback to give the client sequence file an opportunity to modify the database options.
6	Increment the UUT index.	—
7	Call PreUUT callback.	Obtains the UUT serial number from the operator.

Table A-2. Order of Actions the Sequential Process Model Test UUTs Entry Point Performs (Continued)

Action Number	Description	Remarks
8	If no more UUTs, go to Action Number 17.	—
9	Determine the report file pathname.	—
10	Clear information from previous loop iteration.	Discards the previous results and clears the report.
11	Call MainSequence callback.	MainSequence callback in the client sequence file performs tests on the UUT.
12	Call PostUUT callback.	Displays a pass, fail, error, or terminate banner.
13	Call TestReport callback.	Generates a test report for the UUT.
14	Call LogToDatabase callback.	Logs test results to a database for the UUT.
15	Write the UUT report to disk.	Appends an existing file or creates a new file.
16	Loop back to Action Number 6.	—
17	Call PostUUTLoop callback.	Callback in the process model file is empty.

Single Pass

Table A-3 lists the most significant steps of the Single Pass entry point in the Sequential process model, in the order that the Single Pass entry point performs them.

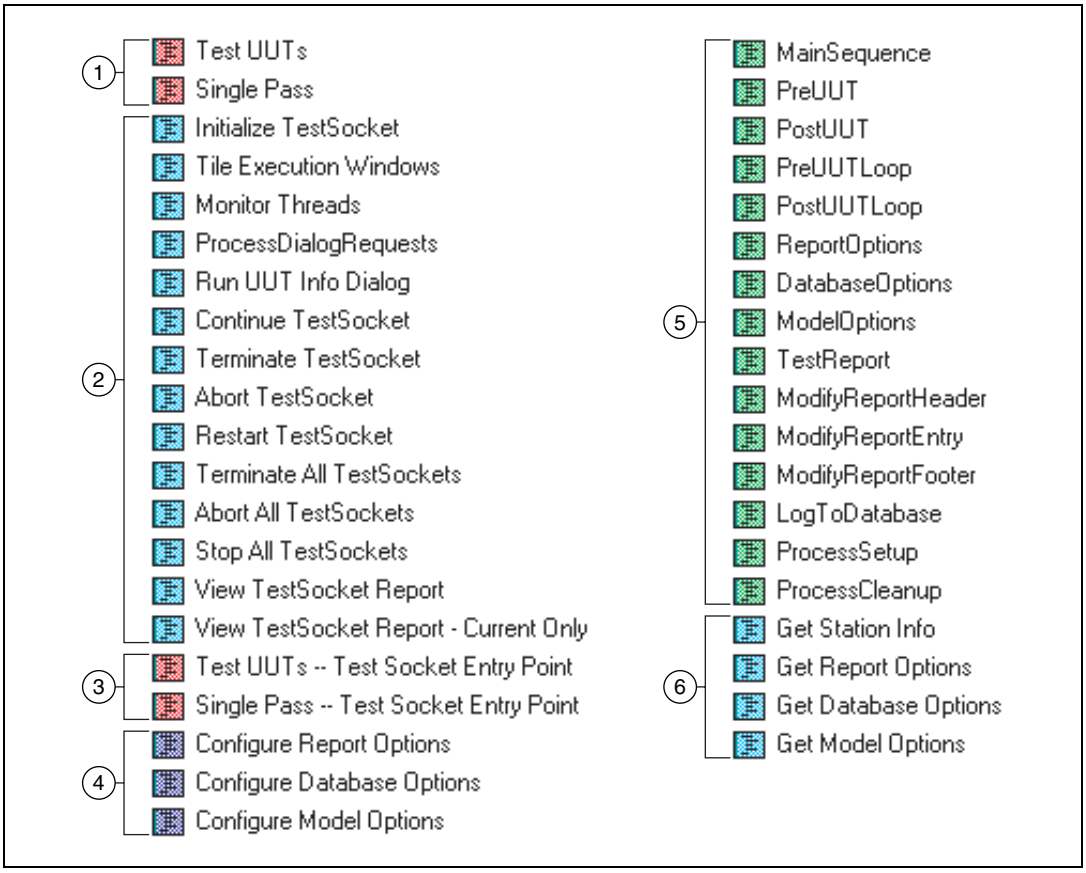
Table A-3. Order of Actions the Sequential Process Model Single Pass Entry Point Performs

Action Number	Description	Remarks
1	Call Get Model Options Utility subsequence.	Reads model options from disk. Calls the ModelOptions callback to give the client sequence file an opportunity to modify the model options.
2	Call Get Station Info Utility subsequence.	Identifies the test station name and the current user.
3	Call Get Report Options Utility subsequence.	Reads report options from disk. Calls the ReportOptions callback to give the client sequence file an opportunity to modify the report options.
4	Call Get Database Options Utility subsequence.	Reads database options from disk. Calls the DatabaseOptions callback to give the client sequence file an opportunity to modify the database options.
5	Determine the report file pathname.	—
6	Call MainSequence callback.	MainSequence callback in the client sequence file performs tests on the UUT.
7	Call TestReport callback.	Generates a test report for the UUT.
8	Write the UUT report to disk.	Appends to an existing file or creates a new file.
9	Call LogToDatabase callback.	Logs test results to a database for the UUT.

Parallel Process Model

Sequences

Figure A-3 shows a list of all the sequences found in the Parallel process model, `ParallelModel.seq`. These sequences are divided into six categories: Execution entry points, Utility sequences, hidden Execution entry points, Configuration entry points, Model callbacks, and Utility subsequences.



1	Main Execution Entry Points	3	Hidden Execution Entry Points	5	Model Callbacks
2	Utility Sequences	4	Configuration Entry Points	6	Utility Subsequences

Figure A-3. Sequences in the Parallel Process Model

Execution Entry Points

The following sequences are the main Execution entry points in the Parallel process model:

- **Test UUTs**—Controls the test socket executions it creates using the Test UUTs – Test Socket Entry Point sequence. When a window for a client sequence file is active, the Test UUTs item is listed in the Execute menu. For more information about the Test UUTs entry point, refer to *Test UUTs* in the *Parallel Process Model* section of this appendix.
- **Single Pass**—Controls the test socket executions it creates using the Single Pass – Test Socket Entry Point sequence. When a window for a client sequence file is active, the Single Pass item is listed in the Execute menu. For more information about the Single Pass entry point, refer to *Single Pass* in the *Parallel Process Model* section of this appendix.

Utility Sequences

The following sequences are Utility sequences in the Parallel process model that are used by the main Execution entry points:

- **Initialize TestSocket**—The controlling execution calls this sequence to initialize the data for and create the test socket executions.
- **Tile Execution Windows**—The controlling execution calls this sequence to tile the test socket Execution windows by building a list of executions and posting a UIMessage to the operator interface requesting it to tile the Execution windows.
- **Monitor Threads**—The ProcessDialogRequests sequence calls this sequence periodically from the controlling execution to poll to see whether any of the test socket executions have been terminated or aborted. If any have, it updates the ModelData for that test socket to indicate its new state, and tells the dialog box to update its display for that test socket.
- **ProcessDialogRequests**—The controlling execution calls this sequence from the Test UUTs sequence. The sequence loops, waiting for requests that the dialog box enqueues into ModelData.DialogRequestQueue. The requests are the names of the sequences to call. When the ProcessDialogRequests sequence receives such a request, it calls the requested sequence. Additionally, this sequence periodically calls the Monitor Threads sequence to verify that the test socket executions are still running and update information about them if they are not.

- **Run UUT Info Dialog**—The controlling execution calls this sequence from a new thread. This sequence initializes and runs the modeless dialog box that the Test UUTs entry point uses to allow the user to control the test socket executions.
- **Continue TestSocket**—Dialog box request callback that the ProcessRequests sequence calls. This sequence sets a notification for the test socket that the request specifies allowing the test socket execution to continue. The test socket execution waits on this notification in its default implementation of the PreUUT and PostUUT callbacks.
- **Terminate TestSocket**—Dialog box request callback that the ProcessDialogRequests sequence calls. The Terminate TestSocket sequence terminates the execution for the test socket that the request specifies.
- **Abort TestSocket**—Dialog box request callback that the ProcessDialogRequests sequence calls. The Abort TestSocket sequence aborts the execution for the test socket that the request specifies.
- **Restart TestSocket**—Dialog box request callback that the ProcessDialogRequests sequence calls. The Restart TestSocket sequence restarts the execution for the test socket that the request specifies. After the sequence restarts the execution, the sequence re-tiles the Execution windows to include the one it restarts.
- **Terminate All TestSocket**—Dialog box request callback that the ProcessDialogRequests sequence calls. The Terminate All Test Socket sequence terminates all of the test socket executions.
- **Abort All TestSocket**—Dialog box request callback that the ProcessDialogRequests sequence calls. The Abort All TestSocket sequence aborts all of the test socket executions.
- **Stop All TestSocket**—Dialog box request callback that the ProcessDialogRequests sequence calls. The Stop All TestSocket sequence sets a flag for each test socket execution telling them to stop after they complete their current UUT test sequence. The sequence also sets a notification to allow them to execute to that point without interruption.

- **View TestSocket Report**—Dialog box request callback that the ProcessDialogRequests sequence calls. The View TestSocket Report sequence launches a report viewer on the report file for the test socket that the request specifies.
- **View TestSocket Report–Current Only**—Dialog box request callback that the ProcessDialogRequests sequence calls. The View TestSocket Report – Current Only sequence launches a report viewer for the report last generated for the test socket that the request specifies. This sequence differs from the View TestSocket Report sequence in that it only shows the last report rather than the whole report file.

Hidden Execution Entry Points

The following sequences are hidden Execution entry points in the Parallel process model, which are used by the main Execution entry points to initiate test socket executions but are never displayed:

- **Test UUTs – Test Socket Entry Point**—The controlling execution uses this entry point to create the test socket executions. If you insert a step into this sequence, disable the Record Results option for the step. The Test UUTs – Test Socket Entry Point sequence implements the Test UUTs process for the test socket executions. For more information about this entry point, refer to *Test UUTs – Test Socket Entry Point* in the *Parallel Process Model* section of this appendix.
- **Single Pass – Test Socket Entry Point**—The controlling execution uses this entry point to create the test socket executions. If you insert a step into this sequence, disable the Record Results option for the step. The Single Pass – Test Socket Entry Point sequence implements the Single Pass process for the test socket executions. For more information about this entry point, refer to *Single Pass – Test Socket Entry Point* in the *Parallel Process Model* section of this appendix.

Configuration Entry Points

The following sequences are Configuration entry points in the Parallel process model:

- **Configure Report Options**, **Configure Database Options**, and **Configure Model Options**—For more information about these sequences, refer to *Configuration Entry Points* in the *Sequential Process Model* section of this appendix.

Model Callbacks

The following sequences are Model callbacks in the Parallel process model, which you can override with a client sequence file:

- **MainSequence**—The Test UUTs – Test Socket Entry Point and Single Pass – Test Socket Entry Point sequences call this callback to test a UUT. The client sequence file must contain a MainSequence callback that performs the tests on a UUT. The MainSequence callback is empty in the process model file.
- **PreUUT**—Calls into the modeless dialog box that the controlling execution creates, which the operator uses to enter UUT serial numbers for the test sockets. The Test UUTs – Test Socket Entry Point sequence calls the PreUUT callback at the beginning of each iteration of the UUT loop. If the operator indicates through the dialog box that no more UUTs are available for testing, the UUT loop terminates. If the operator chooses to stop testing, the code for the dialog box sets the TestSocket.ContinueTesting parameter to `False`. If the operator enters a serial number, the code for the dialog box stores the serial number in the TestSocket.UUT.SerialNumber parameter.
- **PostUUT**—Calls into the modeless dialog box that the controlling execution creates to tell it to display a banner indicating the result of the test that the MainSequence callback in the client sequence file performs on the UUT. The Test UUTs – Test Socket Entry Point calls the PostUUT callback at the end of each iteration of the UUT loop.
- **PreUUTLoop**—The Test UUTs – Test Socket Entry Point sequence calls this callback before the UUT loop begins. The PreUUTLoop callback in the process model file is empty.
- **PostUUTLoop**—The Test UUTs – Test Socket Entry Point sequence calls this callback after the UUT loop terminates. The PostUUTLoop callback in the process model file is empty.
- **ReportOptions**, **DatabaseOptions**, **ModelOptions**, **TestReport**, **ModifyReportHeader**, **ModifyReportEntry**, **ModifyReportFooter**, and **LogToDatabase**—For more information about these sequences, refer to *Model Callbacks* in the *Sequential Process Model* section of this appendix.

- **Process Setup**—The Test UUTs and Single Pass entry points call this callback from the Setup step group to give the client sequence file an opportunity to execute any setup steps that must run only once during the execution of the process model. These setup steps are only run from the controlling execution. The test socket executions do not call this callback.
- **Process Cleanup**—The Test UUTs and Single Pass entry points call this callback from the Cleanup step group to give the client sequence file an opportunity to execute any cleanup steps that must run only once during the execution of the process model. These cleanup steps are only run from the controlling execution. The test socket executions do not call this callback.

Utility Subsequences

The following sequences are Utility subsequences in the Parallel process model, which are called by the other sequences in the Parallel process model:

- **Get Station Info**, **Get Report Options**, **Get Database Options**, and **Get Model Options**—For more information about these sequences, refer to *Utility Subsequences* in the *Sequential Process Model* section of this appendix.

Test UUTs

The Test UUTs entry point is the sequence that the controlling execution runs. Table A-4 lists the most significant steps of the Test UUTs entry point in the order that they are performed.

Table A-4. Order of Actions the Parallel Process Model Test UUTs Entry Point Performs

Action Number	Description	Remarks
1	Call Get Model Options Utility subsequence.	Reads model options from disk. Calls the ModelOptions callback to give the client sequence file an opportunity to modify the model options.
2	Call Get Station Info Utility subsequence.	Identifies the test station name and the current user.
3	Call Get Report Options Utility subsequence.	Reads report options from disk. Calls the ReportOptions callback to give the client sequence file an opportunity to modify the report options.
4	Call Get Database Options Utility subsequence.	Reads database options from disk. Calls the DatabaseOptions callback to give the client sequence file an opportunity to modify the database options.
5	Call Run UUT Info Dialog Utility subsequence.	Creates a modeless dialog box that displays information and gathers serial numbers for the test socket executions.
6	Determine the report file pathname.	Determines the report file pathname to use if the report options are configured so that all UUT results for the model are written to the same file.
7	Create and initialize test socket executions.	For more information about the Test UUTs – Test Socket Entry Point sequence and what test executions do, refer to Table A-5.
8	Call ProcessDialogRequests.	Waits for dialog box requests in a loop until the model is ready to be shut down.

Test UUTs – Test Socket Entry Point

The Test UUTs – Test Socket entry point is the sequence that the test socket executions run. The controlling execution creates the test socket executions in the Test UUTs entry point sequence. Table A-5 lists the most significant steps of the Test UUTs – Test Socket entry point in the order that they are performed.

Table A-5. Order of Actions the Parallel Process Model Test UUTs – Test Socket Entry Point Performs

Action Number	Description	Remarks
1	Call PreUUTLoop callback.	Callback in the process model file is empty.
2	Increment the UUT index.	—
3	Clear information from previous loop iteration.	Discards the previous results and clears the report and failure stacks.
4	Call PreUUT callback.	Obtains the UUT serial number from the operator.
5	If no more UUTs, go to Action Number 13.	—
6	Determine the report file pathname.	—
7	Call MainSequence callback.	MainSequence callback in the client sequence file performs the tests on the UUT.
8	Call PostUUT callback.	Tells the modeless dialog box that the controlling execution creates to display a pass, fail, error, or terminate banner for this test socket.
9	Call TestReport callback.	Generates a test report for the UUT.
10	Call LogToDatabase callback.	Logs test results to a database for the UUT.
11	Write the UUT report to disk.	Appends to an existing file or creates a new file.
12	Loop back to Action Number 2.	—
13	Call PostUUTLoop callback.	Callback in the process model file is empty.

Single Pass Entry Point

The Single Pass entry point is the sequence that the controlling execution runs. Table A-6 lists the most significant steps of the Single Pass entry point in the order that they are performed.

Table A-6. Order of Actions the Parallel Process Model Single Pass Entry Point Performs

Action Number	Description	Remarks
1	Call Get Model Options Utility subsequence.	Reads model options from disk. Calls the ModelOptions callback to give the client sequence file an opportunity to modify the model options.
2	Call Get Station Info Utility subsequence.	Identifies the test station name and the current user.
3	Call Get Report Options Utility subsequence.	Reads report options from disk. Calls the ReportOptions callback to give the client sequence file an opportunity to modify the report options.
4	Call Get Database Options Utility subsequence.	Reads database options from disk. Calls the DatabaseOptions callback to give the client sequence file an opportunity to modify the database options.
5	Determine the report file pathname.	Determines the report file pathname to use if the report options are configured so that all UUT results for the model are written to the same file.
6	Create and initialize test socket executions.	For more information about the Single Pass – Test Socket Entry Point sequence and what test executions do, refer to Table A-7.
7	Wait for test socket executions to complete.	—

Single Pass – Test Socket Entry Point

The Single Pass – Test Socket entry point is the sequence that the test socket executions run. The controlling execution creates the test socket executions in its Single Pass entry point sequence. Table A-7 lists the most significant steps of the Single Pass – Test Socket entry point in the order that they are performed.

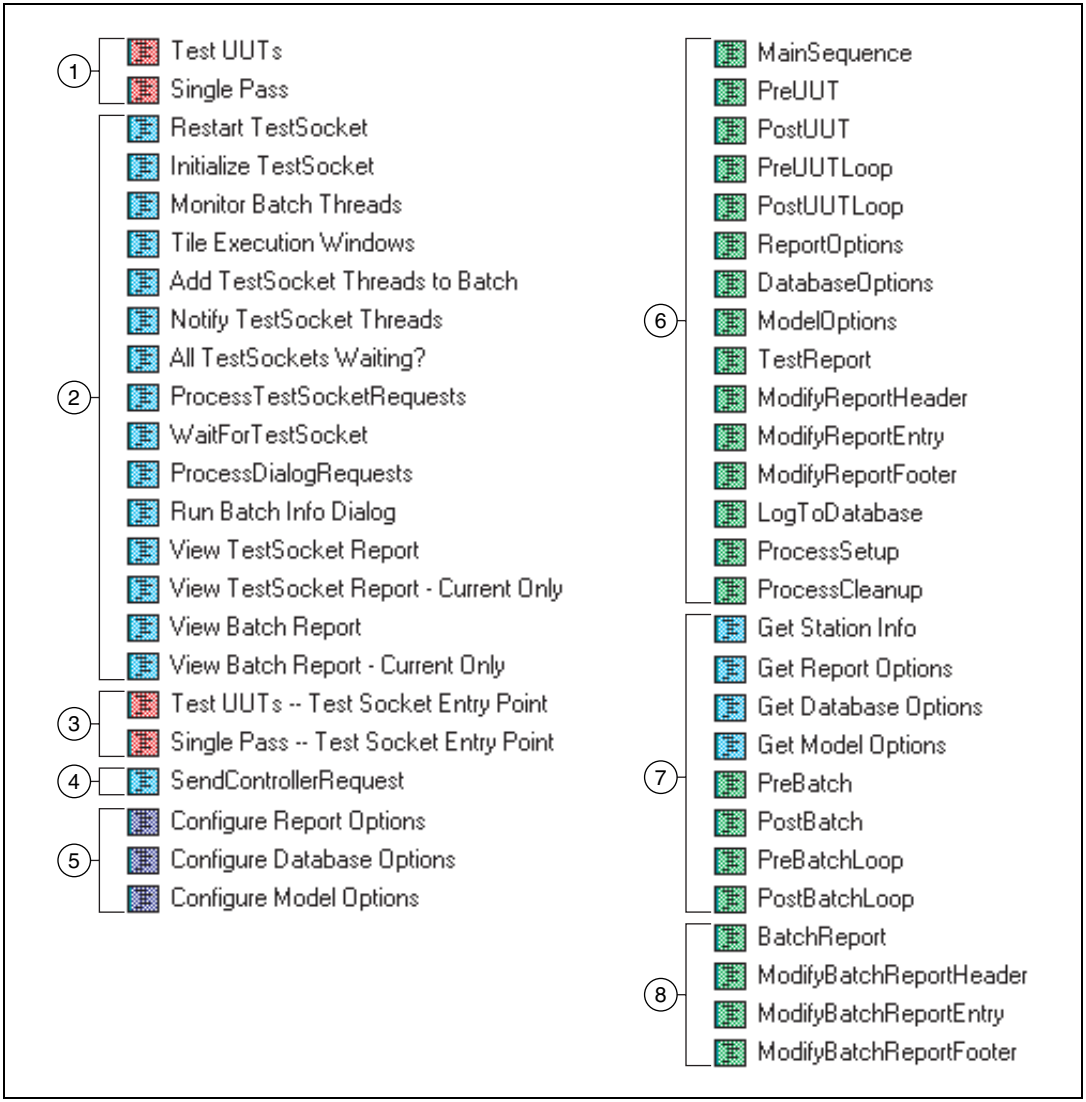
Table A-7. Order of Actions the Parallel Process Model Single Pass – Test Socket Entry Point Performs

Action Number	Description	Remarks
1	Determine the report file pathname.	—
2	Call MainSequence callback.	MainSequence callback in the client sequence file performs the tests on the UUT.
3	Call TestReport callback.	Generates a test report for the UUT.
4	Call LogToDatabase callback.	Logs test results to a database for the UUT.
5	Write the UUT report to disk.	Appends to an existing file or creates a new file.

Batch Process Model

Sequences

Figure A-4 shows a list of all the sequences found in the Batch process model, `BatchModel.seq`. These sequences are divided into the following categories: main Execution entry points, Utility sequences, hidden Execution entry points, Configuration entry points, Model callbacks, and Utility subsequences.



1	Main Execution Entry Points	4	Utility Sequence	7	Utility Subsequences
2	Utility Sequences	5	Configuration Entry Points	8	Model Callbacks
3	Hidden Execution Entry Points	6	Model Callbacks		

Figure A-4. Sequences in the Batch Process Model

Main Execution Entry Points

The following sequences are the main Execution entry points in the Batch process model:

- **Test UUTs**—Runs in the controlling execution of the process model. TestUUTs creates a separate execution for each test socket using the Test UUTs – Test Socket Entry Point sequence, adds the main threads of those executions to a Batch Synchronization object, and controls the flow of execution using queues and notifications such that, all test socket executions execute the Main sequence of the client sequence file together as a group. After a group of UUTs executes, this sequence generates a batch report, loops back around to run the client sequence file on the next group of UUTs, and controls the subsidiary test socket executions to keep them synchronized. When a window for a client sequence file is active, the Test UUTs item is listed in the Execute menu. For more information about the Test UUTs entry point, refer to *Test UUTs* in the *Batch Process Model* section of this appendix.
- **Single Pass**—Runs in the controlling execution of the process model. Singe Pass creates a separate execution for each test socket using the Single Pass–Test Socket Entry Point sequence, adds the main threads of those executions to a Batch Synchronization object, and controls the flow of execution using queues and notifications such that, all test socket executions execute the Main sequence of the client sequence file together as a group. After the group of UUTs executes, this sequence generates a batch report and waits for all subsidiary executions to complete. When a window for a client sequence file is active, the Single Pass item is listed in the Execute menu. For more information about the Single Pass entry point, refer to *Single Pass* in the *Batch Process Model* section of this appendix.

Utility Sequences

The following Utility sequences are used by the main Execution entry points in the Batch process model:

- **Restart TestSocket**—Dialog box request callback that the ProcessDialogRequests sequence calls. This callback restarts the execution for the test socket that the request specifies.
- **Initialize TestSocket**—Called by the controlling execution, initializes the data for and creates the test socket executions.
- **Monitor Batch Threads**—ProcessDialogRequests, ProcessTestSocketRequests, and WaitForTestSocket call this sequence periodically from the controlling execution to poll whether any of the test socket executions have been terminated or aborted. If any have,

Monitor Batch Threads updates the ModelData parameter for that test socket to indicate its new state and tells the dialog box to update its display for that test socket.

- **Tile Execution Windows**—Called by the controlling execution, tiles the test socket Execution windows by building a list of executions and posting a UIMessage to the operator interface, requesting that the operator interface tile the Execution windows. This sequence only tiles running, non-disabled test socket executions.
- **Add TestSocket Threads to Batch**—The Test UUTs and Single Pass entry points call this 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 after running the Main sequence of the client sequence file. Removal from the batch is done in the Test UUTs – Test Socket Entry Point and the Single Pass – Test Socket Entry Point sequences.
- **Notify TestSocket Threads**—The controlling execution calls this sequence to tell the running test socket execution threads to continue executing from their last call to SendControllerRequest in which they block. This sequence optionally waits for each test socket to get to its next call—SendControllerRequest, which is the next synchronization point—before telling the next test socket to go. This ensures serial execution of the test socket executions for the sections of their sequences following the location at which they currently block.
- **All TestSockets Waiting?**—Returns `True` if all running test sockets are waiting for the WaitingForRequest parameter or if all test sockets are stopped.
- **ProcessTestSocketRequests**—The controlling execution calls this sequence to wait for the test socket executions to synchronize at the appropriate point in the execution. When all running test sockets are at the appropriate point in their executions, the sequence returns, allowing the controlling execution to continue. While waiting for the test sockets, this sequence monitors the test socket threads to make sure they are still running. If all test sockets stop running, this sequence will return to allow the controlling sequence to continue.
- **WaitForTestSocket**—The controlling execution calls this sequence from the Notify TestSocket Threads sequence to wait for a test socket execution to receive its next controller request, such as a synchronization point, before the next test socket execution continues. This guarantees that the controlling execution only allows one test socket to run particular sections of its sequence at a time. This sequence is used to write the test socket reports to a file in test socket

index order when the configuration of report options specifies that they are to write reports to the same file.

- **ProcessDialogRequests**—Called by the controlling execution from the Test UUTs sequence. ProcessDialogRequest loops while waiting for requests that the dialog box enqueues into the ModelData.DialogRequestQueue. These requests are the names of the sequences to call. When the ProcessDialogRequests sequence receives a request, it calls the requested sequence. Additionally, this sequence periodically calls the Monitor Batch Threads sequence to make sure that the test socket executions are still running and to update information about them if they are not.
- **Run Batch Info Dialog**—The controlling execution calls this sequence from a new thread in the Test UUTs entry point. The Run Batch Info Dialog sequence initializes and runs the dialog box that allows you to enter serial numbers and view the results for a particular run of the batch.
- **View TestSocket Report**—Dialog box request callback that the ProcessDialogRequests sequence calls. The View TestSocket Report sequence launches a report viewer on the report file for the test socket that the request specifies.
- **View TestSocket Report – Current Only**—Dialog box request callback that the ProcessDialogRequests sequence calls. This sequence launches a report viewer for the last report generated for the test socket that the request specifies. The View TestSocket Report – Current Only sequence differs from the View TestSocket Report sequence in that it only shows the last report, rather than the whole report file.
- **View Batch Report**—Dialog box request callback that the ProcessDialogRequests sequence calls. This sequence launches a report viewer on the report file for the batch report.
- **View Batch Report – Current Only**—Dialog box request callback that the ProcessDialogRequests sequence calls. This callback launches a report viewer for the last batch report generated. The View Batch Report – Current Only sequence differs from the View Batch Report sequence in that it only shows the last report, rather than the whole batch report file.

Hidden Execution Entry Points

The following hidden Execution entry points in the Batch process model are used by the main Execution entry points to start the test socket executions. The hidden Execution entry points are never displayed.

- **Test UUTs – Test Socket Entry Point**—The controlling execution uses the entry point to create the test socket executions. If you insert a step into this sequence, disable the Record Results option for the step. This sequence implements the Test UUTs entry point for the test socket executions. For more information about this entry point, refer to *Test UUTs – Test Socket Entry Point* in the *Batch Process Model* section of this appendix.
- **Single Pass – Test Socket Entry Point**—The controlling execution uses the entry point to create the test socket executions. If you insert a step into this sequence, disable the Record Results option for the step. This sequence implements the Single Pass entry point for the test socket executions. For more information about this entry point, refer to *Single Pass – Test Socket Entry Point* in the *Batch Process Model* section of this appendix.

Utility Sequence

This Utility sequence in the Batch process model is used by the hidden test socket Execution entry points:

- **SendControllerRequest**—The test socket executions call this sequence to synchronize the controlling execution at various locations in their sequences. The test socket executions pass string parameters that indicate the reason and location at which they are attempting to synchronize with the other executions. When all of the test socket executions that are running synchronize with the controlling sequence at the same location by calling the SendControllerRequest sequence, the controlling execution's sequence then performs operations and tells the test socket execution when to continue.

Configuration Entry Points

The following sequences in the Batch process model are the Configuration entry points in the Batch process model:

- **Configure Report Options**, **Configure Database Options**, and **Configure Model Options**—For more information about these sequences, refer to *Configuration Entry Points* in the *Sequential Process Model* section of this appdendix.

Model Callbacks

The following sequences in the Batch process model are Model callbacks, which you can use to override in a client sequence file:

- **MainSequence**—The Test UUTs – Test Socket Entry Point and Single Pass – Test Socket Entry Point sequences call this callback to test a UUT. The client sequence file must contain a MainSequence callback that performs the tests on a UUT. The MainSequence callback is empty in the process model file.
- **PreUUT**—The test socket executions call this callback. The implementation of this sequence is empty in the Batch process model. You can override this callback in the client sequence file to get the serial number for the UUT. If you choose to do this, you should also override the PreBatch callback. In the Batch model, the PreBatch callback displays a dialog box to get the serial numbers for all of the UUTs in the batch. You can find an example illustrating how to override these callbacks in the `<TestStand>\Examples\Callbacks\BatchModel` directory.
- **PostUUT**—The test socket executions call this callback. The implementation of this sequence is empty in the Batch process model. You can override this callback in the client sequence file to display the result status for a UUT. If you choose to do this, you should also override the PostBatch callback. In the Batch model, the PostBatch callback displays a dialog box to show the result status for all of the UUTs in the batch. You can find an example illustrating how to override these callbacks in the `<TestStand>\Examples\Callbacks\BatchModel` directory.
- **PreUUTLoop**—The Test UUTs – Test Socket Entry Point sequence calls this callback before the UUT loop begins. The PreUUTLoop callback in the process model file is empty.
- **PostUUTLoop**—The Test UUTs – Test Socket Entry Point sequence calls this callback after the UUT loop terminates. The PostUUTLoop callback in the process model file is empty.
- **ReportOptions**, **DatabaseOptions**, **ModelOptions**, **TestReport**, **ModifyReportHeader**, **ModifyReportEntry**, **ModifyReportFooter**, and **LogToDatabase**—For more information about these sequences, refer to *Model Callbacks* in the *Sequential Process Model* section of this appendix.
- **Process Setup**—The Test UUTs and Single Pass entry points call this callback from the Setup step group to give the client sequence file an opportunity to execute any setup steps that must run only once during the execution of the process model. These setup steps are run from the

controlling execution only. The test socket executions do not call this callback.

- **Process Cleanup**—The Test UUTs and Single Pass entry points call this callback from the Cleanup step groups to give the client sequence file an opportunity to execute any cleanup steps that must run only once during the execution of the process model. These cleanup steps are run from the controlling execution only. The test socket executions do not call this callback.

Utility Subsequences

The following Utility subsequences in the Batch process model are called by other sequences within the Batch process model:

- **Get Station Info**, **Get Report Options**, **Get Database Options**, and **Get Model Options**—For more information about these sequences, refer to *Utility Subsequences* in the *Sequential Process Model* section of this appendix.
- **PreBatch**—Displays a dialog box in which the operator enters the batch and UUT serial numbers. You can override this in the client sequence file to change or replace this action. You can find an example illustrating how to override this callback in the `<TestStand>\Examples\Callbacks\BatchModel` directory.
- **PostBatch**—Displays a pass, fail, error, or terminated banner for each test socket and allows viewing of batch and UUT reports. You can override this callback in the client sequence file to change or replace this action. You can find an example illustrating how to override this callback in the `<TestStand>\Examples\Callbacks\BatchModel` directory.
- **PreBatchLoop**—The process model calls this callback before looping on a batch of UUTs. This callback is empty in the process model file. You can override this callback in the client sequence file to perform an action before the batch is tested.
- **PostBatchLoop**—The process model calls this callback after looping on a batch of UUTs. This callback is empty in the process model file. Override this callback in the client sequence file to perform an action after all batches of UUTs are tested.

Model Callbacks

The following sequences are Model callbacks in the Batch process model that are unique to the model and called by the main Execution entry points:

- **BatchReport**—The Test UUTs and Single Pass entry points call this callback to generate the contents of the batch report for the UUTs that ran in the last batch. You can override the BatchReport callback in the client sequence file if you want to change its behavior entirely. The Batch process model defines a batch report for a single group of UUTs as a header, an entry for each UUT result, and a footer. If you do not override the BatchReport callback, you can override the ModifyBatchReportHeader, ModifyBatchReportEntry, and ModifyBatchReportFooter callbacks to customize the batch report.
- **ModifyBatchReportHeader**—The BatchReport callback calls this callback so that the client sequence file can modify the batch report header. ModifyBatchReportHeader receives the following parameters: 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 calls this callback so that the client sequence file can modify the entry for each test socket's UUT result in the batch report. Using subsequences, the BatchReport callback calls the ModifyBatchReportEntry callback for each test socket. The ModifyBatchReportEntry callback receives the following parameters: the test socket data, the tentative report entry text, and the report options. The ModifyBatchReportEntry callback in the process model file is empty.
- **ModifyBatchReportFooter**—The BatchReport callback calls this callback so that the client sequence file can modify the batch report footer. The ModifyBatchReportFooter callback receives the following parameters: the tentative report footer text and the report options. The ModifyBatchReportFooter callback in the process model file is empty.

Test UUTs

The Test UUTs entry point is the sequence that the controlling execution runs. Table A-8 lists the most significant steps of the Test UUTs entry point in the order that they are performed.

Table A-8. Order of Actions the Batch Process Model Test UUTs Entry Point Performs

Action Number	Description	Remarks
1	Call Get Model Options Utility subsequence.	Reads model options from disk. Calls the ModelOptions callback to give the client sequence file an opportunity to modify the model options.
2	Call PreBatchLoop callback.	Callback in the process model file is empty.
3	Call Get Station Info Utility subsequence.	Identifies the test station name and the current user.
4	Call Get Report Options Utility subsequence.	Reads report options from disk. Calls the ReportOptions callback to give the client sequence file an opportunity to modify the report options.
5	Call Get Database Options Utility subsequence.	Reads database options from disk. Calls the DatabaseOptions model callback to give the client sequence file an opportunity to modify the database options.
6	Create and initialize test socket executions.	For more information about the Test UUTs – Test Socket Entry Point sequence and what test executions do, refer to Table A-9.
7	Call Run Batch Info Dialog.	Calls the Run Batch Info Dialog sequence in a new thread and waits for it to initialize the dialog box code.
8	Wait for test sockets to get to the Initialize synchronization point.	Calls the ProcessTestSocketRequests sequence to wait for and monitor test socket executions.
9	Call Add TestSocket Threads to Batch.	Adds test socket execution threads to the Batch Synchronization object. This allows the user's test sequence to use batch synchronization.
10	Allow test socket executions that are waiting at the Initialize synchronization point to continue.	Calls the Notify TestSocket Threads sequence.
11	Increment the Batch index.	—

Table A-8. Order of Actions the Batch Process Model Test UUTs Entry Point Performs (Continued)

Action Number	Description	Remarks
12	Wait for test sockets to get to the GetUUTSerialNumber synchronization point.	Calls the ProcessTestSocketRequests sequence to wait for and monitor test socket executions.
13	Call PreBatch callback.	Obtains the batch and UUT serial numbers from the operator.
14	If no more UUTs, set test socket data to tell test sockets to stop running their UUT loops.	Sets the ContinueTesting test socket data variable to `False` for all of the test sockets and marks them all as enabled so that they will be added to the batch and exit normally.
15	Remove disabled test socket threads from the batch and add enabled test socket threads.	Disabled test sockets need to be removed from the batch so that they don't block the threads that are running.
16	Allow test socket executions that are waiting at the GetUUTSerialNumber synchronization point to continue.	Calls the Notify TestSocket Threads sequence.
17	If no more UUTs, go to Action Number 34.	—
18	Wait for test sockets to get to the ReadyToRun synchronization point.	Calls the ProcessTestSocketRequests sequence to wait for and monitor test socket executions.
19	Determine the report file pathname for Batch and UUT report files.	Determines the report file pathname to use if the report options are configured so that all UUT results for the model are written to the same file or are written to the same file as the batch reports.
20	Allow test socket executions that are waiting at the ReadyToRun synchronization point to continue.	Calls the Notify TestSocket Threads sequence.
21	Wait for test sockets to get to the ShowStatus synchronization point.	Calls the ProcessTestSocketRequests sequence to wait for and monitor test socket executions.

Table A-8. Order of Actions the Batch Process Model Test UUTs Entry Point Performs (Continued)

Action Number	Description	Remarks
22	Call Add TestSocket Threads to Batch.	The test socket executions remove themselves from the batch after executing MainSequence in order to cleanup the state of the batch in case the sequence was terminated or the user did not match enters and exits properly. This is where the test socket execution threads are added to the batch again.
23	Call PostBatch callback.	Displays a pass, fail, error, or terminate banner for all of the test sockets in the batch.
24	Allow test socket executions that are waiting at the ShowStatus synchronization point to continue.	Calls the Notify TestSocket Threads sequence.
25	Wait for test sockets to get to the WriteReport synchronization point.	Calls the ProcessTestSocketRequests sequence to wait for and monitor test socket executions.
26	Call BatchReport callback.	Generates a batch report for the last run of the batch of UUTs.
27	Write the batch report to disk.	Appends to an existing file or creates a new file.
28	Allow test socket executions that are waiting at the WriteReport synchronization point to continue.	Calls the Notify TestSocket Threads sequence passing `True` for the ReleaseThreadsSequentially parameter so that only one UUT report is written at a time in the test socket index order.
29	Wait for test sockets to get to the UUTDone synchronization point.	Calls the ProcessTestSocketRequests sequence to wait for and monitor test socket executions.
30	Tell the Status dialog box that report generation is complete.	Enables the View Report button so that you can view the reports from the dialog box.
31	Wait for Status dialog box.	If the PostBatch callback Status dialog box displays the PostBatch callback, then the sequence waits for you to dismiss the dialog box, if you have not already done so.

Table A-8. Order of Actions the Batch Process Model Test UUTs Entry Point Performs (Continued)

Action Number	Description	Remarks
32	Allow test socket executions that are waiting at the UUTDone synchronization point to continue.	Calls the Notify TestSocket Threads sequence.
33	Loop back to Action Number 11.	—
34	Wait for test socket executions to complete.	—
35	Call PostBatchLoop callback.	Callback in the process model file is empty.

Test UUTs – Test Socket Entry Point

The Test UUTs – Test Socket entry point is the sequence that the test socket executions run. The controlling execution creates the test socket executions in its Test UUTs entry point sequence. Table A-9 lists the most significant steps of the Test UUTs – Test Socket entry point in the order that they are performed.

Table A-9. Order of Actions the Batch Process Model Test UUTs – Test Socket Entry Point Performs

Action Number	Description	Remarks
1	Synchronize with the controlling execution for the Initialize synchronization point.	Calls SendControllerRequest and blocks until the controlling execution sets the test socket's notification.
2	Call PreUUTLoop callback.	Callback in the process model file is empty.
3	Increment the UUT index.	—
4	Clear information from previous loop iteration.	Discards the previous results and clears the report and failure stack.
5	Synchronize with the controlling execution for the GetUUTSerialNumber synchronization point.	Calls SendControllerRequest and blocks until the controlling execution sets the test socket's notification.
6	Call PreUUT callback.	Callback in the process model file is empty.
7	If no more UUTs, go to Action Number 20.	—

Table A-9. Order of Actions the Batch Process Model Test UUTs – Test Socket Entry Point Performs (Continued)

Action Number	Description	Remarks
8	Synchronize with the controlling execution for the ReadyToRun synchronization point.	Calls SendControllerRequest and blocks until the controlling execution sets the test socket's notification.
9	Determine the report file pathname.	—
10	Call MainSequence callback.	MainSequence callback in the client sequence file performs the tests on the UUT.
11	Remove the test socket thread from batch synchronization.	Cleans the state of the batch in case the MainSequence was terminated or you did not match enters and exits properly. The controlling execution adds the thread to batch synchronization before continuing past the next synchronization point. Disabled test sockets do not get added to the batch.
12	Synchronize with the controlling execution for the ShowStatus synchronization point.	Calls SendControllerRequest and blocks until the controlling execution sets the test socket notification.
13	Call PostUUT callback.	Callback in the process model file is empty.
14	Call TestReport callback.	Generates a test report for the UUT.
15	Call LogToDatabase callback.	Logs test results to a database for the UUT.
16	Synchronize with controlling execution for the WriteReport synchronization point.	Calls SendControllerRequest and blocks until the controlling execution sets the test socket notification.
17	Write the UUT report to disk.	Appends to an existing file or creates a new file.
18	Synchronize with controlling execution for the UUTDone synchronization point.	Calls SendControllerRequest and blocks until the controlling execution sets the test socket notification.
19	Loop back to Action Number 3.	—
20	Call PostUUTLoop callback.	Callback in the process model file is empty.

Single Pass

The Single Pass entry point is the sequence that the controlling execution runs. Table A-10 lists the most significant steps of the Single Pass entry point in the order that they are performed.

Table A-10. Order of Actions the Batch Process Model Single Pass Entry Point Performs

Action Number	Description	Remarks
1	Call Get Model Options Utility subsequence.	Reads model options from disk. Calls the ModelOptions callback to give the client sequence file an opportunity to modify the model options.
2	Call Get Station Info Utility subsequence.	Identifies the test station name and the current user.
3	Call Get Report Options Utility subsequence.	Reads report options from disk. Calls the ReportOptions callback to give the client sequence file an opportunity to modify the report options.
4	Call Get Database Options Utility subsequence.	Reads database options from disk. Calls the DatabaseOptions callback to give the client sequence file an opportunity to modify the database options.
5	Create and initialize test socket executions.	For more information about the Single Pass – Test Socket Entry Point sequence and what test executions do, refer to Table A-11.
6	Wait for test sockets to get to the ReadyToRun synchronization point.	Calls the ProcessTestSocketRequests sequence to wait for and monitor test socket executions.
7	Call Add TestSocket Threads to Batch.	Adds test socket execution threads to the batch Synchronization object. This allows your test sequence to use batch synchronization.
8	Determine the report file pathname for batch and UUT report files.	Determines the report file pathname to use if the report options are configured so that all UUT results for the model are written to the same file or if they are written to the same file as the batch reports.

Table A-10. Order of Actions the Batch Process Model Single Pass Entry Point Performs (Continued)

Action Number	Description	Remarks
9	Allow test socket executions that are waiting at the ReadyToRun synchronization point to continue.	Calls the Notify TestSocket Threads sequence.
10	Wait for test sockets to get to the PostMainSequence synchronization point.	Calls the ProcessTestSocketRequests sequence to wait for and monitor test socket executions.
11	Call Add TestSocket Threads to Batch.	The test socket executions remove themselves from the batch after executing MainSequence, in order to clean up the state of the batch in case the sequence was terminated or you did not match enters and exits properly. This is where the test socket execution threads are added to the batch again.
12	Allow test socket executions that are waiting at the PostMainSequence synchronization point to continue.	Calls the Notify TestSocket Threads sequence.
13	Wait for test sockets to get to the WriteReport synchronization point.	Calls the ProcessTestSocketRequests sequence to wait for and monitor test socket executions.
14	Call BatchReport callback.	Generates a batch report for the last batch of UUTs run.
15	Write the batch report to disk.	Appends to an existing file or creates a new file.
16	Allow test socket executions that are waiting at the WriteReport synchronization point to continue.	Calls the Notify TestSocket Threads sequence passing `True` for the ReleaseThreadsSequentially parameter so that only one UUT report is written at a time in test socket index order.
17	Wait for test sockets to get to the UUTDone synchronization point.	Calls the ProcessTestSocketRequests sequence to wait for and monitor test socket executions.

Table A-10. Order of Actions the Batch Process Model Single Pass Entry Point Performs (Continued)

Action Number	Description	Remarks
18	Allow test socket executions that are waiting at the UUTDone synchronization point to continue.	Calls the Notify TestSocket Threads sequence.
19	Wait for test socket executions to complete.	—

Single Pass – Test Socket Entry Point

The Single Pass –T est Socket entry point is the sequence that the test socket executions run. The controlling execution creates the test socket executions in its Single Pass entry point sequence. Table A-11 lists the most significant steps of the Single Pass – Test Socket entry point in the order that they are performed.

Table A-11. Order of Actions the Batch Process Model Single Pass – Test Socket Entry Point Performs

Action Number	Description	Remarks
1	Sync with controlling execution for the ReadyToRun synchronization point.	Calls SendControllerRequest and blocks until the controlling execution sets the test socket's notification.
2	Determine the report file pathname.	—
3	Call MainSequence callback.	MainSequence callback in the client sequence file performs the tests on the UUT.
4	Remove the test socket thread from batch synchronization.	Allows other test socket threads to do batch synchronization without counting this thread anymore. The controlling execution adds the thread to batch synchronization before the thread runs the Main sequence again. Disabled test sockets do not get added to the batch.
5	Synchronize with controlling execution for the PostMainSequence synchronization point.	Calls SendControllerRequest and blocks until the controlling execution sets the test socket's notification.
6	Call TestReport callback.	Generates a test report for the UUT.

Table A-11. Order of Actions the Batch Process Model Single Pass – Test Socket Entry Point Performs (Continued)

Action Number	Description	Remarks
7	Call LogToDatabase callback.	Logs test results to a database for the UUT.
8	Synchronize with controlling execution for the WriteReport synchronization point.	Calls SendControllerRequest and blocks until the controlling execution sets the test socket notification.
9	Write the UUT report to disk.	Appends to an existing file or creates a new file.
10	Synchronize with controlling execution for the UUTDone synchronization point.	Calls SendControllerRequest and blocks until the controlling execution sets the test socket's notification.

Support Files for the TestStand Process Models

Many sequences in the TestStand process model files call functions in DLLs and subsequences in other sequence files. TestStand installs these supporting files and the DLL source files in the same directory that it installs the process model sequence files.

Table A-12 lists the supporting files that TestStand installs for the TestStand process models in the `<TestStand>\Components\NI\Models\TestStandModels` directory.

Table A-12. Installed Support Files for the Process Model Files

File Name	Description
`SequentialModel.seq`, `ParallelModel.seq`, and `BatchModel.seq`	Entry point and Model callback sequences for the TestStand process models.
`reportgen_html.seq`	Subsequences that add the header, result entries, and footer for a UUT into an HTML test report.
`reportgen_txt.seq`	Subsequences that add the header, result entries, and footer for a UUT into an ASCII-text test report.
`reportgen_xml.seq`	Subsequences that add the deader, result entries, and footer for a UUT into an XML test report.

Table A-12. Installed Support Files for the Process Model Files (Continued)

File Name	Description
modelsupport2.dll	DLL containing C functions that the process model sequences call. Includes functions that launch the Report Options and Model Options dialog boxes, read and write those options from disk, determine the report file pathname, obtain the UUT serial number from the operator, and display status banners.
modelsupport2.prj	LabWindows/CVI project that builds modelsupport2.dll.
modelsupport2.fp	LabWindows/CVI function panels for the functions in modelsupport2.dll.
modelsupport2.h	C header file that contains declarations for the functions in modelsupport2.dll.
modelsupport2.lib	Import library in Visual C/C++ format for modelsupport2.dll.
modelpanels.uir	LabWindows/CVI user interface resource file containing panels that the functions in modelsupport2.dll use.
ModelSupport.seq	Subsequences that all process models use for report generation.
database.seq	Subsequences that all process models use for database logging.
modelpanels.h	C header file containing declarations for the panels in modelpanels.uir.
main.c	C source for utility functions.
banners.c	C source for functions that display status banners.
report.c	C source for functions that launch the Report Options dialog box, read and write the report options from disk, and determine the report file pathname.
uutdlg.c	C source for the function that obtains the UUT serial number from the operator.
c_report.c	C source for generating HTML, XML, and ASCII-text reports for the DLL option in the Report Options dialog box.

Table A-12. Installed Support Files for the Process Model Files (Continued)

File Name	Description
modeloptions.c	C source for the functions that launch the Model Options dialog box and read and write the model options from disk.
batchUUTdlg.c and parallelUUTdlg.c	C source for the functions that launch the UUT identification dialog boxes for the Batch and Parallel process models. The files are part of modelsupport2.dll but the default process model, SequentialModel.seq, does not call them.

You can view the contents of the reportgen_html.seq, reportgen_txt.seq, and reportgen_xml.seq sequence files in the sequence editor. These files are model sequence files and contain an empty ModifyReportEntry callback. Each file has a PutOneResultInReport sequence that calls ModifyReportEntry. The client sequence file can override the ModifyReportEntry callback. TestStand requires that all sequence files that contain direct calls to Model callbacks must also contain a definition of the callback sequence and must be model files.

The TestStand process model sequence files also contain an empty ModifyReportEntry callback, even though no sequences in those files call ModifyReportEntry directly. They contain a ModifyReportEntry callback so that ModifyReportEntry appears in the Sequence File Callbacks dialog box for the client sequence file.

Report Generation Functions and Sequences

When you customize report generation for your test station, create your own process model, or modify the default TestStand process model files, always make a copy of the default process model and then make your modifications to that copy. This practice ensures that newer installations of TestStand will not overwrite your customizations. Place the copy in the <TestStand>\Components\User directory.

Tables A-13 and A-14 list the process model sequences and C functions that generate the report and the locations of the files that contain them. Table A-13 lists the default process model sequences in the <TestStand>\Components\NI\Models\TestStandModels directory that generate the report header and footer.

Table A-13. Sequences that Generate the Report Header and Footer

Report Format	Report Header or Footer	
	Header	**Footer**
HTML	AddReportHeader sequence in `reportgen_html.seq`	AddReportFooter sequence in `reportgen_html.seq`
Text	AddReportHeader sequence in `reportgen_txt.seq`	AddReportFooter sequence in `reportgen_txt.seq`
XML	AddReportHeader sequence in `reportgen_xml.seq`	AddReportHeader sequence in `reportgen_xml.seq`

Table A-14 lists the default process model sequences and C functions in `<TestStand>\Components\NI\Models\TestStandModels` that generate the report body for each step result.

Table A-14. Sequences or C Functions that Generate the Report Body

Report Format	Report Body Generator Selected in the Report Options Dialog Box	
	Sequence	**DLL**
HTML	PutOneResultInReport sequence in `reportgen_html.seq`	PutOneResultInReport_Html function in `c_report.c` in the `modelsupport2.prj` LabWindows/CVI project.
Text	PutOneResultInReport sequence in `reportgen_txt.seq`	PutOneResultInReport_Txt function in `c_report.c` in the `modelsupport2.prj` LabWindows/CVI project.
XML	AddReportBody sequence in `reportgen_xml.seq`	AddSequenceCallResult_XML function in the `modelsupport2.prj` LabWindows/CVI project.

You can also alter the report generation for each client sequence file that you run. To alter report generation, you override the report generation Model callbacks in the client sequence file. Table A-15 lists the report generation Model callbacks.

Table A-15. Report Generation Model Callbacks

Section of the Report to Alter	Model Callback Sequence to Override
Report Header	ModifyReportHeader
Report Footer	ModifyReportFooter
Each Step Result	ModifyReportEntry (TestStand does not call this callback if you select DLL in the Select a Report Generator for Producing the Report Body section of the Contents tab on the Report Options dialog box.)
Entire Report	TestReport

In addition, each step in the sequence can add text to its corresponding result in the report. To make these additions, the step stores the text to add to the report in its Step.Result.ReportText property.

B

Synchronization Step Types

This appendix describes step types that you use to synchronize, pass data between, and perform other operations in multiple threads of an execution or multiple running executions in the same process. Configure these steps using the Configuration dialog boxes. Do not write code modules for these steps.

For more information about the Configuration dialog boxes for Synchronization step types, refer to the *TestStand Help*. You can view examples for Synchronization step types in the `<TestStand>\Examples\Synchronization` directory.

Synchronization Objects

Most Synchronization step types create and control a particular type of Synchronization object. Following is a list of the types of Synchronization objects:

- **Lock**—Use a Lock object to guarantee exclusive access to a resource. For example, if several execution threads write to a device that does not have a thread-safe driver, you can use a Lock object to make sure that only one thread accesses the device at a time.
- **Semaphore**—Use a Semaphore object to limit access to a resource to a specific number of threads. A Semaphore object is similar to a Lock object, except that it restricts access to the number of threads that you specify rather than to just one thread. For example, you can use a Semaphore object to restrict access to a communications channel to a limited number of threads so that each thread has sufficient bandwidth. Typically, you limit access to a shared resource to only one thread at a time. Therefore, a typical application uses Lock objects rather than Semaphore objects.
- **Rendezvous**—Use a Rendezvous object to make a specific number of threads wait for each other before they proceed past a location you specify. For example, if different threads configure different aspects of a testing environment, you can use a Rendezvous object to ensure that none of the threads proceed beyond the configuration process until all threads have completed their configuration tasks.

- **Queue**—Use a Queue object to pass data from the thread that produces it to a thread that processes it. For example, a thread that performs tests asynchronously with respect to the Main sequence might use a queue to receive commands from the Main sequence.
- **Notification**—Use a Notification object to notify one or more threads when a particular event or condition occurs. For example, if you display a dialog box in a separate thread, you can use a Notification object to signal another thread when the user dismisses the dialog box.
- **Batch**—Use a Batch object to define and synchronize a group of threads. This is useful when you want to test a group of similar UUTs simultaneously. You can configure a synchronized section so that only one UUT enters the section at a time, no UUTs enter the section until all are ready, and no UUTs proceed beyond the section until all are done. This is useful when, for a particular test, you only have one test resource which you must apply to each UUT in turn. You can also configure a synchronized section to guarantee that only one thread executes the steps in the section. This is useful for an action that applies to the entire batch, such as raising the temperature in an environmental chamber. Having a separate thread for each UUT allows you to exploit parallelism while enforcing serialization when necessary. It also allows you to use preconditions and other branching options so that each UUT has its own flow of execution.

  Normally, you are not required to create a Batch object. The TestStand Batch process model does this for you. The model uses Batch Specification steps to group test socket execution threads together so that you can use Batch Synchronization steps to synchronize them in your sequence file. If you want to create a synchronized section around a single step, use the Synchronization tab on the Step Properties dialog box instead of using explicit Batch Synchronization steps.

  For more information about the Batch process model, refer to the *Batch Process Model* section of Appendix A, *Process Model Architecture*. For more information about Batch Synchronization, refer to the *Batch Synchronization* section of this appendix. For more information about the Synchronization tab on the Step Properties dialog box, refer to the *TestStand Help*.

Common Attributes of Synchronization Objects

You can use the Configuration dialog box for each step type to configure the following attributes for all Synchronization objects:

Name

When you create a Synchronization object, you can specify a unique name with a literal string or an expression that evaluates to a string. If an object with the same name and type already exists, you create a reference to the existing object. Otherwise, you create a reference to a new Synchronization object. By creating a reference to an existing object, you can access the same Synchronization object from multiple threads or executions.

If you specify an empty string as the name for a Synchronization object, TestStand creates an unnamed Synchronization object that you can only access through an object reference variable. To associate an unnamed Synchronization object with an object reference variable, select **Use Object Reference** as the object reference lifetime in the <StepType> Step Configuration dialog box for each step type.

By default, a Synchronization object is only accessible from the operating system process in which you create it. However, you can make a Synchronization object accessible from other processes, such as multiple instances of an operator interface, by using an asterisk (`*`) as the first character in the name. In addition, you can create a Synchronization object on a specific machine by beginning the name with the machine name, such as `"\\\\machinename\\syncobjectname"`. You can then use this name to access the Synchronization object from any machine on your network.

To access Synchronization objects on other machines, you must configure DCOM for the `TSAutoMgr.exe` server, which is located in the `<TestStand>\Bin` directory. Refer to the *Setting up TestStand as a Server for Remote Execution* section of Chapter 5, *Module Adapters*, for information about configuring DCOM and setting up TestStand as a server for remote execution. Follow the instructions given for the `REngine.exe` server, but apply them to the `TSAutoMgr.exe` server.

Note When you specify an object on a remote machine using a string constant in a dialog box expression control, be sure to escape the backslashes and surround the name in quotes. For example, use `"\\\\machinename\\syncobjname"` instead of `\\machinename\\syncobjname`.

All named TestStand Synchronization objects share the same name space. Therefore, you cannot have Synchronization objects with the same name. Synchronization object names are not case-sensitive.

Lifetime

When you create a Synchronization object, you must specify a lifetime for the reference you create. The object exists for at least as long as the reference exists, but can exist longer if another reference to it has a different lifetime.

The object reference lifetime choices are Same as Sequence, Same as Thread, Same as Execution, and Use Object Reference. If you refer to your object by name only, then you typically set its reference lifetime to Same as Sequence, Same as Thread, or Same as Execution. This guarantees that the object lives as long as the sequence, thread, or execution in which you create the reference. If you want to explicitly control the lifetime of the object reference or if you wish to refer to the object using an object reference variable, select Use Object Reference from the <Step> Reference Lifetime ring control in the <Step Type> Step Configuration dialog box. You can use the object reference in place of its name when performing operations on the object.

You can also use the reference from other threads without performing a Create operation in each thread. An object reference releases its object when you set the variable equal to `Nothing`, when you reuse the variable to store a different reference, or when the variable goes out of scope. When the last object reference to a Synchronization object releases, TestStand disposes of the object.

Some Synchronization objects have an operation, such as Lock or Acquire, for which you can also specify a lifetime. In this case, the lifetime determines the duration of the operation.

Timeout

Most Synchronization objects can perform one or more operations that timeout if they do not complete within the number of seconds you specify. You can specify that TestStand treats a timeout as an error condition or you can explicitly check for the occurrence of a timeout by checking the value of the Step.Result.TimeoutOccurred property.

Synchronization Step Types

Each type of Synchronization object has a step type to create and control the object. The Batch Synchronization object has two step types, Batch Specification and Batch Synchronization. For all other Synchronization objects, the name of the step type is the same as the name of the Synchronization object type it controls. The following additional Synchronization step types exist:

- **Wait**—Use the Wait step to wait for an execution or thread to complete or for a time interval to elapse.
- **Thread Priority**—Use the Thread Priority step to adjust the operating system priority of a TestStand thread.

To use any Synchronization step type, insert a step of that type and select **Configure <Step Name>** from the context menu to launch the <Step Type> Step Configuration dialog box. Use this dialog box to select an operation for the step to perform. You can then specify settings for the operation you select. Some operations store output values to variables you specify. If the control for an output value is labeled as an optional output, you can leave the control empty.

The following sections describe the functionality and custom properties of each Synchronization step type.

Lock

Use a Lock step to ensure that only one thread can access a particular resource or data item at a time. For example, if you examine and update the value of a global variable from multiple threads or executions, you can use a lock to ensure that only one thread examines and updates the variable at a time. If multiple threads are waiting to lock a lock, they do so in first in first out (FIFO) order as the lock becomes available.

A thread can lock the same lock an unlimited number of times without unlocking it. To release the lock, the thread must balance each Lock operation with an Unlock operation.

Locks in TestStand have deadlock detection. If all of the threads that are using a set of locks reside on the same machine, and all of the locks in that set reside on that machine as well, TestStand will detect and report a run-time error if deadlock occurs as a result of those locks and threads. To avoid deadlock, you must always lock a set of locks in the same order in every thread, or lock all of the locks required by a thread in one Lock operation by specifying an array of lock names or references.

Note You can also create a lock around a single step using the Synchronization tab on the Step Properties dialog box.

Note TestStand variables and properties are thread safe.

Step Properties

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

- **Step.Result.TimeoutOccurred**—Set to `True` if the Lock operation times out. This property only exists if the step is configured for the Lock operation.
- **Step.NameOrRefExpr**—Contains the Lock Name expression for the Create operation and the Lock Name or Reference expression for all other Lock operations. In the case of the Lock operation, this expression can also specify an array of names or references.
- **Step.LifetimeRefExpr**—Contains the object reference expression for the Lock Reference Lifetime or Lock Operation Lifetime when you set either lifetime to Use Object Reference.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Lock operation.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Lock operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Lock operation.
- **Step.AlreadyExistsExpr**—Contains the Already Exists expression for the Create operation or the Lock Exists expression for the Get Status operation.
- **Step.NumThreadsWaitingExpr**—Contains the Number of Threads Waiting to Lock the Lock expression for the Get Status operation.
- **Step.Operation**—Contains a value that specifies the operation the step is configured to perform. The valid values are `0` = Create, `1` = Lock, `2` = Early Unlock, `3` = Get Status.
- **Step.Lifetime**—Contains a value that specifies the lifetime setting to use for the Create operation. The valid values are `0` = Same as Sequence, `1` = Same as Thread, `2` = Use Object Reference, `3` = Same as Execution.

- **Step.LockLifetime**—Contains a value that specifies the lifetime setting to use for the Lock operation. The valid values are `0` = Same as Sequence, `1` = Same as Thread, `2` = Use Object Reference.
- **Step.CreateIfDoesNotExist**—Contains the Create If Does Not Exist setting for the Lock operation.

Rendezvous

Use a Rendezvous step to cause threads to wait for each other before proceeding past a specified location. Each thread blocks as it performs the Rendezvous operation. When the number of blocked threads reaches the total that you specified when you created the rendezvous, the rendezvous unblocks all its waiting threads and they resume execution.

Step Properties

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

- **Step.Result.TimeoutOccurred**—Set to `True` if the Rendezvous operation times out. This property only exists if the step is configured for the Rendezvous operation.
- **Step.NameOrRefExpr**—Contains the Rendezvous Name expression for the Create operation and the Rendezvous Name or Reference expression for other Rendezvous operations.
- **Step.LifetimeRefExpr**—Contains the object reference expression for the Rendezvous Reference lifetime when you set the lifetime to Use Object Reference.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Rendezvous operation.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Rendezvous operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Rendezvous operation.
- **Step.AlreadyExistsExpr**—Contains the Already Exists expression for the Create operation or the Rendezvous Exists expression for the Get Status operation.
- **Step.RendezvousCountExpr**—Contains the Number of Threads Per Rendezvous expression for the Create operation.
- **Step.NumThreadsWaitingExpr**—Contains the Number of Threads Waiting for Rendezvous expression for the Get Status operation.

- **Step.Operation**—Contains a value that specifies the operation the step performs. The valid values are `0` = Create, `1` = Rendezvous, `2` = Get Status.
- **Step.Lifetime**—Contains a value that specifies the lifetime for the Create operation. The valid values are `0` = Same as Sequence, `1` = Same as Thread, `2` = Use Object Reference, `3` = Same as Execution.
- **Step.RendezvousCountOutExpr**—Contains the Number of Threads Per Rendezvous expression for the Get Status operation.

Queue

Use Queue steps to synchronize the production and consumption of data among your threads. A queue has two primary operations—enqueue and dequeue. Enqueue places a data item on the queue and dequeue removes an item from the queue. The Enqueue operation blocks when the queue is full, while the Dequeue operation blocks when the queue is empty. If multiple threads block on the same Queue operation, the threads unblock in first in first out (FIFO) order.

Step Properties

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

- **Step.Result.TimeoutOccurred**—Set to `True` if an Enqueue or Dequeue operation times out. This property only exists if the step is configured for the Enqueue or Dequeue operation.
- **Step.NameOrRefExpr**—Contains the Queue Name expression for the Create operation and the Queue Name or Reference expression for all other operations. In the case of the Dequeue operation, this expression can specify an array of names or references.
- **Step.LifetimeRefExpr**—Contains the object reference expression for the queue lifetime when you set the lifetime to Use Object Reference.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Enqueue or Dequeue operation.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Enqueue or Dequeue operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Enqueue or Dequeue operation.

- **Step.AlreadyExistsExpr**—Contains the Already Exists expression for the Create operation or the Queue Exists expression for the Get Status operation.
- **Step.MaxNumElementsExpr**—Contains the expression that specifies the maximum number of elements of the queue for the Create operation.
- **Step.MaxNumElementsOutExpr**—Contains the expression that specifies where to store the maximum number of elements of the queue for the Get Status operation.
- **Step.NumThreadsWaitingEnqueueExpr**—Contains the expression that specifies where to store the number of threads that are waiting to enqueue for the Get Status operation.
- **Step.NumThreadsWaitingDequeueExpr**—Contains the expression that specifies where to store the number of threads that are waiting to dequeue for the Get Status operation.
- **Step.Operation**—Contains a value that specifies the operation the step performs. The valid values are `0` = Create, `1` = Enqueue, `2` = Dequeue, `3` = Flush, `4` = Get Status.
- **Step.Lifetime**—Contains a value that specifies the lifetime setting for the Create operation. The valid values are `0` = Same as Sequence, `1` = Same as Thread, `2` = Use Object Reference, `3` = Same as Execution.
- **Step.NumElementsExpr**—Contains the expression that specifies where to store the current number of elements in the queue for the Get Status operation.
- **Step.DataExpr**—Contains the New Element to Enqueue expression when you configure the step for the Enqueue operation, the Location to Store Element expression when you configure the step for the Dequeue operation, and the Location to Store Array of Queue Elements expression when you configure the step for the Flush or Get Status operation.
- **Step.ByRef**—Contains the Boolean value that specifies whether the step stores a queue element by object reference instead of by value for the Enqueue operation.
- **Step.EnqueueLocation**—Contains a value that specifies the location to store the queue element for the Enqueue operation. The valid values are `0` = Front of Queue, `1` = Back of Queue.
- **Step.DequeueLocation**—Contains a value that specifies the location to remove the queue element from for the Dequeue operation. The valid values are `0` = Front of Queue, `1` = Back of Queue.

- **Step.FullQueueOption**—Contains a value that specifies the options for the If the Queue is Full setting of the Enqueue operation. The valid values are `0` = Wait, `1` = Discard Front Element, `2` = Discard Back Element, `3` = Do Not Enqueue.
- **Step.RemoveElement**—Contains a Boolean value that specifies whether the step removes the element from the queue when it performs the Dequeue operation.
- **Step.WhichQueueExpr**—Contains the expression that specifies where to store the array offset of the queue on which the Dequeue operation occurs.

Notification

Use Notification steps to notify one or more threads when a particular event or condition has been met. You can also pass data to the threads you notify.

Step Properties

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

- **Step.Result.TimeoutOccurred**—Set to `True` if a Wait operation times out. This property only exists if the step is configured for the Wait operation.
- **Step.NameOrRefExpr**—Contains the Notification Name expression for the Create operation and the Notification Name or Reference expression for all other operations. In the case of the Wait operation, this expression can optionally specify an array of names or references.
- **Step.LifetimeRefExpr**—Contains the object reference expression for the notification lifetime when you set the lifetime to Use Object Reference.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Wait operation.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Wait operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Wait operation.
- **Step.AlreadyExistsExpr**—Contains the Already Exists expression for the Create operation or the Notification Exists expression for the Get Status operation.

- **Step.NumThreadsWaitingExpr**—Contains the expression that specifies where to store the number of threads that are waiting on the notification for the Get Status operation.
- **Step.Operation**—Contains a value that specifies the operation the step is set to perform. The valid values are `0` = Create, `1` = Set, `2` = Clear, `3` = Pulse, `4` = Wait, `5` = Get Status.
- **Step.Lifetime**—Contains a value that specifies the lifetime setting for the Create operation. The valid values are `0` = Same as Sequence, `1` = Same as Thread, `2` = Use Object Reference, `3` = Same as Execution.
- **Step.DataExpr**—Contains the Data Value expression for the Set or Pulse operation, or the Location to Store Data expression for the Wait or Get Status operation.
- **Step.ByRef**—Contains the Boolean value that specifies whether to store the data by object reference instead of by value for a Set or Pulse operation.
- **Step.WhichNotificationExpr**—Contains the expression that specifies where to store the array offset of the notification to which the Wait operation responds.
- **Step.IsSetExpr**—Contains the expression that specifies where to store the Boolean value that indicates whether the notification is in a Set state. The Get Status operation uses this expression.
- **Step.IsAutoClearExpr**—Contains the expression that specifies where to store the Boolean value that indicates whether the notification is configured to AutoClear. The Get Status operation uses this expression.
- **Step.AutoClear**—Contains the AutoClear setting for the Set operation.
- **Step.PulseNotifyOpt**—Contains the setting for the Pulse operation that indicates the threads to which a pulse notification is sent. The valid values are `0` = Notify First Waiting Thread, `1` = Notify All Waiting Threads.

Wait

Use Wait steps to wait for an execution or thread to complete or for a time interval to elapse.

Retrieving the Results from Executions and Threads

When the thread or execution completes, the Wait step copies the result status and error information for the thread or execution to its own status and error properties. Therefore, if a Wait step waits on a sequence that fails, the status of the Wait step is `Failed`.

The result list entry for a Wait step contains a TS.AsyncSequenceCall.ResultList property which is the result list for the thread or execution. You can also access the same result list in the TS.SequenceCall.ResultList property in the result for the Sequence Call step that launches the thread or execution.

Step Properties

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

- **Step.Result.TimeoutOccurred**—Set to `True` if the Wait for Thread or Wait for Execution operation times out. This property only exists if the step is configured for one of these operations.
- **Step.TimeoutEnabled**—Contains the timeout enabled setting for the Wait for Thread or the Wait for Execution operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Wait for Thread or the Wait for Execution operation.
- **Step.ThreadRefExpr**—Contains the Thread Reference expression for the Wait for Thread operation when the Step.SpecifyBySeqCall property is set to `False`.
- **Step.SeqCallName**—Contains the name of the Sequence Call step that creates the thread or execution the step waits for when the Step.SpecifyBySeqCall property is set to `True`.
- **Step.SeqCallStepGroupIdx**—Contains the step group of the Sequence Call step that creates the thread or execution that the step waits for when the Step.SpecifyBySeqCall property is set to `True`. The valid values are `0` = Setup, `1` = Main, `2` = Cleanup.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Wait for Thread or the Wait for Execution operation.
- **Step.WaitForTarget**—Contains a value that specifies the type of Wait operation the step performs. The valid values are `0` = Time Interval, `1` = Time Multiple, `2` = Thread, `3` = Execution.
- **Step.TimeExpr**—Contains the time expression for the Time Interval or Time Multiple operation of the step.

- **Step.ExecutionRefExpr**—Contains the expression that evaluates to a reference to the execution on which the Wait for Execution operation waits.
- **Step.SpecifyBySeqCall**—Contains the Specify By Sequence Call setting for the Wait for Thread or the Wait for Execution operation.

TestStand also adds the following properties to the results for Wait steps that are configured to wait for a thread or execution.

Note These properties are not subproperties of the Result property for the Wait step type.

- **AsyncMode**—Set to `True` if the Wait step is waiting on a thread. It is set to `False` if the Wait step is waiting on an execution.
- **AsyncId**—Contains the value of the Id property of the thread or execution that the step is waiting for.

Batch Synchronization

Use Batch Synchronization steps to define sections of a sequence in which to synchronize multiple threads that belong to one batch. Typically, you use these steps in a sequence that you execute using the Batch process model.

More specifically, you place Batch Synchronization steps around one or more test steps to create a synchronized section.

Synchronized Sections

Use Batch Synchronization steps to define synchronized sections by placing a step at the beginning and end of a section of steps in a sequence and specifying an Enter operation for the beginning step and an Exit operation for the ending step. While you must place the Enter and Exit steps in the same sequence, you do not have to place them in the same step group. There are three types of synchronized sections—Serial, Parallel, and One Thread Only. All synchronized sections share the following behaviors:

- Each thread in a batch that enters a synchronized section blocks at the Enter step until all other threads in the batch arrive at their respective instances of the Enter step.
- Each thread in a batch that reaches the end of the synchronized section blocks at the Exit step until all other threads in the batch arrive at their respective instances of the Exit step.

Serial Sections

Use a Serial section to ensure that each thread in the batch executes the steps in the section sequentially and in the order that 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 add them to the batch using the Batch Specification step. 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, they exit the section together. Refer to the *Semaphore* section of this appendix for more information about order numbers.

Parallel Sections

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. Each thread that arrives at the Exit step for the section blocks until all threads in the batch reach that step.

One Thread Only Sections

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 the thread with the lowest order number. 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. The threads in the batch then exit the section together.

Mismatched Sections

Sections become mismatched when all threads in a batch are blocked at an Enter or an Exit operation, but they are not all blocked at the same Enter or Exit operation. This can occur when a sequence has a conditional flow of execution due to preconditions, post actions, or other flow control operations.

When TestStand detects mismatched sections, it handles them as follows:

- The thread that is at the Enter or Exit step that appears earliest in the hierarchy of sequences and subsequences proceeds as if all threads in the batch are at the same step.
- If multiple Enter and Exit operations are equally early in the hierarchy of sequences and subsequences, Enter operations proceed first.

Nested Sections

Nesting of sections can occur either within the same sequence or as a result of calling a subsequence inside of a synchronized section when the subsequence also contains a synchronized section. When you nest one section inside another, TestStand honors the inner section if the type of the outer section is serial or parallel. For example, if you nest one serial section in another serial section, each thread that enters the outer section proceeds only until the Enter step of the inner section and then waits for the other threads to reach the same step.

TestStand ignores the inner section if the type of the outer section is One Thread Only.

Note You can create a synchronized section around a single step using the **Synchronization** tab on the Step Properties dialog box rather than by using explicit Batch Synchronization steps.

Requirements for Using Enter and Exit Operations

TestStand generates a run-time error if your Enter and Exit operations do not adhere to the following requirements:

- Each Exit operation must match the most nested Enter operation.
- A thread cannot reenter a section it is already within.
- You must exit a section in the same sequence that you enter it.

Step Properties

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

- **Step.Result.TimeoutOccurred**—Set to `True` if an Enter or Exit operation times out.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Enter or Exit operation.

- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Enter or Exit operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Enter or Exit operation.
- **Step.Operation**—Contains a value that specifies the operation the step performs. The valid values are `0` = Enter Synchronized Section, `1` = Exit Synchronized Section.
- **Step.SectionNameExpr**—Contains the expression that specifies the name of the section for the Enter or Exit operation.
- **Step.SectionType**—Contains a value that specifies the type of section the Enter operation defines. The valid values are `1` = Serial, `2` = Parallel, `3` = OneThreadOnly.

Thread Priority

Use the Thread Priority step to increment or decrement the priority of a thread so that it receives more or less CPU time than other threads.

When you use this step, you must avoid starving important threads of CPU time by boosting the priority of another thread too high. When you alter a thread priority, remember to save the previous priority value and restore it once your thread no longer requires the altered priority value.

Note Setting the priority of a thread to **Time Critical** can cause the user interface for your application to become unresponsive.

Step Properties

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

- **Step.Operation**—Contains a value that specifies the operation the step is set to perform. The valid values are `0` = Set Thread Priority, `1` = Get Thread Priority.
- **Step.SetPriorityExpr**—Specifies the thread priority expression for the Set Thread Priority operation.
- **Step.GetPriorityExpr**—Specifies the location to store the thread priority for the Get Thread Priority operation.

Semaphore

Use Semaphore steps to limit concurrent access to a resource to a specific number of threads. A semaphore stores a numeric count and allows threads to increment (release) or decrement (acquire) the count as long as the count stays equal to or greater than zero. If a decrement would cause the count to go below zero, the thread attempting to decrement the count blocks until the count increases. When multiple threads are waiting to decrement a semaphore, the semaphore unblocks the threads in first in first out (FIFO) order whenever another thread increments the semaphore count.

A semaphore with an initial count of one behaves like a lock, with one exception. Like a lock, a one-count semaphore restricts access to a single thread at a time. Unlike a lock, a thread cannot acquire a one-count semaphore multiple times without first releasing it after each acquire. When a thread attempts to acquire the semaphore a second time without releasing it, the count is zero and the thread blocks. Refer to the *Lock* section of this appendix for more information about Lock objects.

Step Properties

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

- **Step.Result.TimeoutOccurred**—Set to `True` if the Acquire operation times out. This property only exists if the step is configured for the Acquire operation.
- **Step.NameOrRefExpr**—Contains the Semaphore Name expression for the Create operation and the Semaphore Name or Reference expression for all of the other operations.
- **Step.AutoRelease**—Contains a Boolean value that specifies whether the Acquire operation automatically performs a release when the Acquire lifetime expires.
- **Step.LifetimeRefExpr**—Contains the object reference expression for the semaphore lifetime or acquire lifetime when you set either lifetime to Use Object Reference.
- **Step.TimeoutEnabled**—Contains the Timeout Enabled setting for the Acquire operation.
- **Step.TimeoutExpr**—Contains the Timeout expression, in seconds, for the Acquire operation.
- **Step.ErrorOnTimeout**—Contains the Timeout Causes Run-Time Error setting for the Acquire operation.

- **Step.AlreadyExistsExpr**—Contains the Already Exists expression for the Create operation or the Semaphore Exists expression for the Get Status operation.
- **Step.InitialCountExpr**—Contains the Numeric expression that the Create operation uses for the initial count of the semaphore.
- **Step.NumThreadsWaitingExpr**—Contains the Number of Threads Waiting to Acquire the Semaphore expression for the Get Status operation. Step.Operation contains a value that specifies the operation the step performs. The valid values are `0` = Create, `1` = Acquire, `2` = Release, `3` = Get Status.
- **Step.Lifetime**—Contains a value that specifies the Lifetime setting for the Create operation. The valid values are `0` = Same as Sequence, `1` = Same as Thread, `2` = Use Object Reference, `3` = Same as Execution.
- **Step.InitialCountOutExpr**—Contains the Initial Semaphore Count expression for the Get Status operation.
- **Step.AcquireLifetime**—Contains a value that specifies the lifetime setting for the Acquire operation. The valid values are `0` = Same as Sequence, `1` = Same as Thread, `2` = Use Object Reference. The Acquire operation only uses this setting when Step.AutoRelease is set to `True`.
- **Step.CurrentCountExpr**—Contains the Current Count expression for the Get Status operation.

Batch Specification

When you write a process model, you can use Batch Specification steps to define a group of threads where each thread in the group runs an instance of the client sequence file. Defining a group allows you to perform Batch Synchronization operations on the threads in the group. The TestStand Batch process model uses Batch Specification steps to create a batch that contains a thread for each test socket. For more information about the Batch process model refer to the *Batch Process Model* section of Appendix A, *Process Model Architecture*. For more information about batch synchronization, refer to the *Batch Synchronization* section of this appendix.

When you test each UUT in a separate thread, you use the Batch Specification step to include the UUT threads in one batch. Use the Batch Synchronization step to control the interaction of the UUT threads as they execute the test steps.

Step Properties

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

- **Step.Operation**—Contains a value that specifies the operation the step performs. The valid values are `0` = Create, `1` = Add Thread, `2` = Remove Thread, `3` = Get Status.
- **Step.NameOrRefExpr**—Contains the Name expression for the Create operation and the Name or Reference expression for other batch operations.
- **Step.Lifetime**—Contains a value that specifies the lifetime for the Create operation. The valid values are `0` = Same as Sequence, `1` = Same as Thread, `2` = Use Object Reference, `3` = Same as Execution.
- **Step.LifetimeRefExpr**—Contains the object reference expression for the batch lifetime when you set the lifetime to Use Object Reference.
- **Step.AlreadyExistsExpr**—Contains the Already Exists expression for the Create operation or the Batch Exists expression for the Get Status operation.
- **Step.ThreadRefExpr**—Contains the Object Reference to Thread expression for the Add Thread and Remove Thread operations.
- **Step.OrderNumExpr**—Contains the Order Number expression for the Add Thread operation.
- **Step.NumThreadsWaitingExpr**—Contains the Number of Threads Waiting at Synchronized Sections expression for the Get Status operation.
- **Step.NumThreadsInBatchExpr**—Contains the Number of Threads in Batch expression for the Get Status operation.
- **Step.DefaultBatchSyncExpr**—Contains the Default Batch Synchronization expression for the Create operation.
- **Step.DefaultBatchSyncOutExpr**—Contains the Default Batch Synchronization expression for the Get Status operation.

C

IVI Step Types

TestStand provides several step types that enable you to configure and acquire data from Interchangeable Virtual Instrument (IVI) class-compliant instruments. IVI is an instrument driver standard that provides common programming interfaces for several classes of instruments. IVI drivers exist for a number of popular instruments, including all applicable devices from National Instruments. For more information about IVI and IVI class-compliant instrument drivers, refer to the National Instruments Web site at `ni.com/ivi`.

You can view examples of the IVI step types in the `<TestStand>\Examples\IVI` directory.

TestStand includes the following IVI step types:

- **Dmm**—Performs single-point and multipoint measurements with digital multimeters.
- **Scope**—Performs single-point and waveform measurements with oscilloscopes.
- **Fgen**—Generates predefined or custom waveforms using arbitrary waveform generators.
- **Power Supply**—Controls and monitors the output of DC power supplies.
- **Switch**—Connects or disconnects paths and routes, determines the connectivity of two switches or the state of a route, and queries the state of the switch module or virtual device.
- **Tools**—Sets or gets instrument attributes and performs utility operations on any IVI instrument.

IVI step types offer a configuration-based approach to instrument control. Use an initial step to configure an instrument, and then perform measurements in one or more subsequent steps. TestStand references a session to an instrument using the instrument logical name that you configure in National Instruments Measurement & Automation Explorer (MAX). TestStand automatically initializes the instrument session when the instrument is first configured and automatically closes the instrument session when the execution is closed. If two executions reference the same

logical name, the session is shared and the session closes when the last execution is released.

IVI step types complement, but do not replace, the instrument configuration and measurement operations you perform in code modules that you write using LabVIEW, Measurement Studio, Microsoft Visual Basic, or other tools. Although IVI step types are the easiest way to configure and acquire data from IVI class instruments, you must use code modules to control instruments under the following circumstances:

- When you need to precisely specify the instrument driver calls to ensure optimal performance.
- When you need to call specific driver functions that an IVI class does not support.
- When your instrument does not conform to an IVI class or does not have an IVI driver.
- When you need to interleave your instrument control operations with other code that must reside in a single code module.

Note TestStand does not install IVI-compliant instrument drivers or configure sample logical names in MAX. Refer to the application help for the IVI Instruments category in MAX for more information about where to obtain and how to install IVI-compliant instrument drivers, as well as information about how to create logical names that the IVI step types use.

Editing an IVI Step

To use an IVI step, insert an IVI step for the class of instrument you want to control. To edit the step, right-click the step and select **Edit <Step Name>** from the context menu. Figure C-1 shows the Configure operation for the IVI Dmm step.

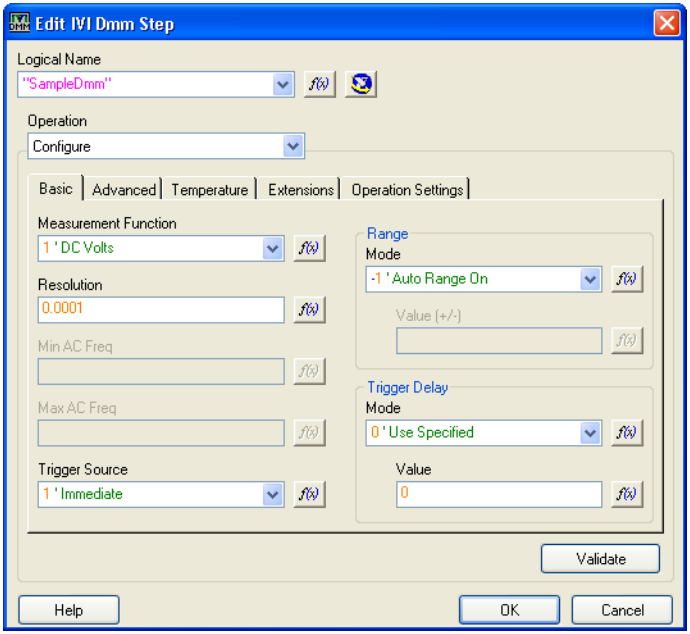


Figure C-1. IVI Dmm Step Configure Operation

Each Edit <Step Name> Step dialog box contains a Logical Name ring control, which you use to select a logical name or a virtual instrument name that you configure in MAX. Use the buttons to the right of the Logical Name ring control to launch the Expression Browser dialog box and to launch MAX.

Note All IVI names, such as logical names, virtual names, and channel names, are case-sensitive. If you use logical names, driver session names, or virtual names in your program, be sure that the name you use exactly matches the name in the IVI Configuration Store file. Do not make any variations in the case of the characters in the name.

The Edit <Step Name> Step dialog box also contains an Operation ring control, which specifies the action the step performs. Typical operations include configuring the instrument, taking a reading, or showing/hiding a graphical display panel for the instrument, also called a soft front panel (SFP). Depending on the instrument, you can also select other low-level actions such as Initiate, Send Software Trigger, or Get Information.

Note When you select an operation, the area under the Operation ring control changes. Many operations group their settings on tab controls.

In some cases, when TestStand configures an instrument, the instrument driver may coerce a settings value. Configuring an instrument might result in an invalid value error for a particular setting because the instrument-based values are not checked for validity until the configuration actually occurs. Once configure completes successfully, you can issue all of the other operations in subsequent steps.

Refer to the *TestStand Help* for more information about the Edit <Step Name> Step dialog box associated with each IVI step type.

Extensions

The Configure operation configures the instrument to match the settings as specified by the step. To enable instrument configuration controls that apply to features that IVI defines as class extensions, select from the Extensions tab the extended features that your instrument supports. Figure C-2 shows the Extensions tab for the IVI Dmm step.

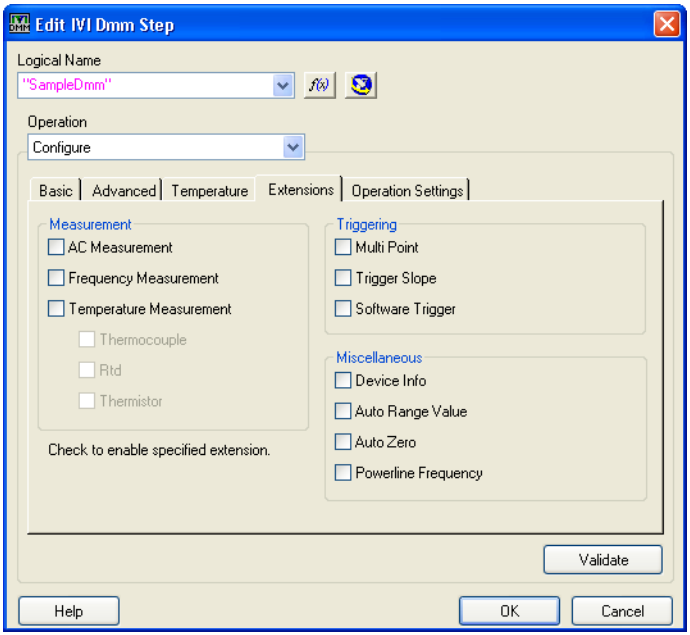


Figure C-2. IVI Dmm Extensions Tab

The Configure operation only handles those settings that are supported by the base class specification and the extension groups specified on the Extensions tab. For the best results, only enable those extensions that are required for your application.

Operation Settings

Many of the operations, such as Configure and Fetch, allow you to specify where the dialog box saves the Operation settings for the step. For example, you could save the Configure settings in a shared variable so that multiple steps could use the same settings. Figure C-3 shows the Operation Settings tab for the Configure operation.

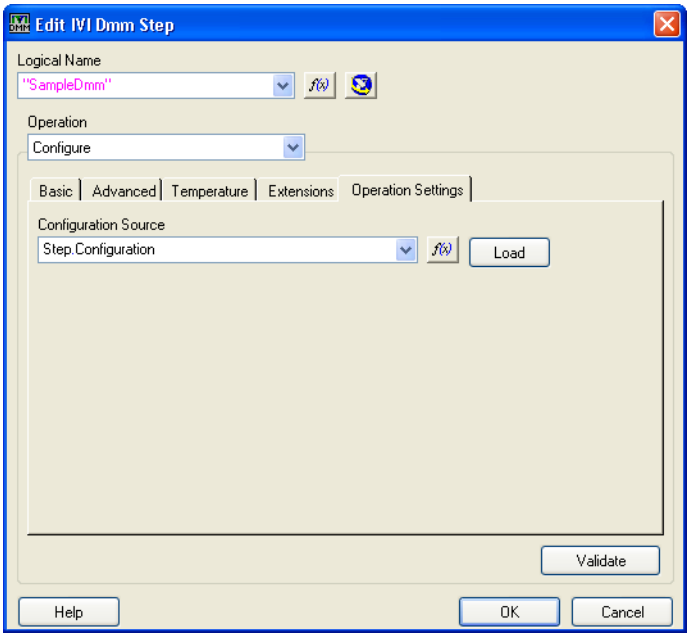


Figure C-3. IVI Dmm Operation Settings Tab

The Configuration Source ring control specifies the name of the property or variable where TestStand stores the settings when you click OK. The Load button reloads the settings from the specified property or variable location.

Validating a Configuration

When you edit a step that configures an instrument, click **Validate** to test your configuration before closing the Edit <Step Name> Step dialog box. Refer to the *TestStand Help* for more information about the Validate IVI Configuration dialog box.

Using Soft Front Panels

Each IVI session for an IVI step type can display a graphical display panel for the instrument, also called a soft front panel (SFP). The Show and Hide Soft Front Panel operations control whether TestStand displays a SFP for the instrument.

When the SFP is visible, you can interact directly with the instrument session that TestStand is controlling.

Refer to the *TestStand Help* for more information about the Show and Hide Soft Front Panel operations.

Get Information

Use the Get Information operation for each instrument class step type to retrieve low-level status and information from the instrument. For the Get Information operation, you must specify an expression that contains a variable or property to which the step assigns the retrieved value. In some cases, you must specify a channel name for the value to retrieve.

Instrument Session Manager

IVI step types use a software component called Session Manager to share named instrument connections. Use Session Manager to share instrument connections in code modules that you write, even if you do not use IVI step types. Refer to the *Session Manager Help* for more information by selecting **Start»National Instruments»Session Manager»NI Session Manager Help**.

Note Currently available drivers do not allow you to use the same instrument driver session in more than one operating system process simultaneously.

IVI Dmm

Use the IVI Dmm step type to perform single-point and multipoint measurements with digital multimeters.

Step Operations

The IVI Dmm step type supports the following operations:

- **Configure**—Configures the instrument to match the state as specified by the step.
- **Show Soft Front Panel**—Displays the SFP for the instrument.
- **Hide Soft Front Panel**—Hides the SFP for the instrument.
- **Read**—Initiates and returns a measurement from an instrument.
- **Initiate**—Initiates a measurement.
- **Fetch**—Returns the measured value from a measurement that the Initiate operation has started.
- **Abort**—Cancels the wait for a trigger.
- **Send SW Trigger**—Sends a software trigger command to trigger the instrument.
- **Get Information**—Retrieves low-level status and information from the instrument.

Refer to the *TestStand Help* for more information about each of these operations.

Step Properties

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

- **Step.Result.Reading**—Contains the measurement values for the Read and Fetch operations. The property data type is either NI_IviSinglePoint or NI_IviWave.
- **Step.LogicalName**—Contains the logical name expression.
- **Step.InstrOperation**—Contains a value that specifies the operation the step is set to perform.
- **Step.SettingsSource**—Contains the name of the property or variable where the step loads and stores the settings for the operation.
- **Step.Configuration**—Contains the settings for the Configure operation. The data type of this property is NI_IviDmmConfig.

- **Step.SoftFrontPanel**—Contains the settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.
- **Step.Readings**—Contains the settings for the Read and Fetch operations.
- **Step.GetInfo**—Contains the settings for the Get Information operation.

IVI Scope

Use the IVI Scope step type to acquire a voltage waveform from an analog input signal with oscilloscopes.

Step Operations

The IVI Scope step type supports the following operations:

- **Configure**—Configures the instrument to match the state as specified by the step.
- **Show Soft Front Panel**—Displays the SFP for the instrument.
- **Hide Soft Front Panel**—Hides the SFP for the instrument.
- **Read**—Initiates and returns a measurement from an instrument.
- **Initiate**—Initiates a measurement.
- **Fetch**—Returns the measured value from a measurement that the Initiate operation has started.
- **Abort**—Cancels the wait for a trigger.
- **Auto Setup**—Performs an automatic setup on the instrument.
- **Get Information**—Retrieves low-level status and information from the instrument.

Refer to the *TestStand Help* for more information about each of these operations.

Step Properties

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

- **Step.Result.Reading**—Contains the measurement values for the Read and Fetch operations. This property is an array of container, and the size of the array is equal to the number of channels specified for the Read or Fetch operation. The data type of each element of the array is NI_IviSinglePoint, NI_IviWave, or NI_IviWavePair.
- **Step.LogicalName**—Contains the logical name expression.
- **Step.InstrOperation**—Contains a value that specifies the operation the step is set to perform.
- **Step.SettingsSource**—Contains the name of the property or variable where the step loads and stores the settings for the operation.
- **Step.Configuration**—Contains the settings for the Configure operation. The data type of this property is NI_IviDmmConfig.
- **Step.SoftFrontPanel**—Contains the settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.
- **Step.Readings**—Contains the settings for the Read and Fetch operations. The data type of this property is NI_IviScopeReadings. The Channels subproperty is an array of type NI_IviReading.
- **Step.GetInfo**—Contains the settings for the Get Information operation.

IVI Fgen

Use the IVI Fgen step type to instruct function generators to generate predefined waveforms or custom waveforms using arbitrary waveform generators.

Step Operations

The IVI Fgen step type supports the following operations:

- **Configure**—Configures the instrument to match the state as specified by the step.
- **Show Soft Front Panel**—Displays the SFP for the instrument.
- **Hide Soft Front Panel**—Hides the SFP for the instrument.
- **Initiate**—Initiates signal generation if the instrument is idle.

- **Abort**—Aborts a previously configured output and returns the function generator to the idle state.
- **Send SW Trigger**—Sends a software trigger command to trigger the instrument.
- **Get Information**—Retrieves low-level status and information from the instrument.

Refer to the *TestStand Help* for more information about each of these operations.

Step Properties

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

- **Step.LogicalName**—Contains the logical name expression.
- **Step.InstrOperation**—Contains a value that specifies the operation the step is set to perform.
- **Step.SettingsSource**—Contains the name of the property or variable where the step loads and stores the settings for the operation.
- **Step.Configuration**—Contains the settings for the Configure operation. The data type of this property is NI_IviFgenConfig.
- **Step.SoftFrontPanel**—Contains the settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.
- **Step.GetInfo**—Contains the settings for the Get Information operation.

IVI Power Supply

Use the IVI Power Supply step type to instruct power supplies to control the output voltages and currents, and measure output values at the output terminals.

Step Operations

The IVI Power Supply step type supports the following operations:

- **Configure**—Configures the instrument to match the state as specified by the step.
- **Show Soft Front Panel**—Displays the SFP for the instrument.
- **Hide Soft Front Panel**—Hides the SFP for the instrument.

- **Measure**—Takes a measurement on the output signal and returns the measured value.
- **Initiate**—Makes the power supply wait for a trigger.
- **Abort**—Cancels the wait for a trigger.
- **Send SW Trigger**—Sends a software trigger command to trigger the instrument.
- **Reset Output Protection**—Resets the power supply's output protection on a specific channel after an overvoltage or overcurrent condition occurs.
- **Get Information**—Retrieves low-level status and information from the instrument.

Refer to the *TestStand Help* for more information about each of these operations.

Step Properties

In addition to the common custom properties, the IVI Power Supply step type defines the following step properties:

- **Step.Result.Reading**—Contains the measurement values for the Measure operation. The property data type is an array of NI_IviSinglePoint.
- **Step.LogicalName**—Contains the logical name expression.
- **Step.InstrOperation**—Contains a value that specifies the operation the step is set to perform.
- **Step.SettingsSource**—Contains the name of the property or variable where the step loads and stores the settings for the operation.
- **Step.SoftFrontPanel**—Contains the settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.
- **Step.Readings**—Contains the settings for the Measure operation.
- **Step.GetInfo**—Contains the settings for the Get Information operation.
- **Step.ResetOutputProtection**—Contains the channel setting for the Reset Output Protection operation.

IVI Switch

The IVI Switch step type provides a high-level programming layer for instruments that are compliant with the IVI Switch class and National Instruments Switch Executive virtual devices. A switch is an instrument that can establish a connection between two I/O channels. The IVI Switch step type also supports IVI-compliant instruments that can perform trigger scanning, and trigger-synchronized establishing or breaking of the paths.

The IVI Switch step type allows you to connect and disconnect paths or routes, determine the connectivity of two switches or the state of a route, and query the state of the switch module or virtual device.

National Instruments Switch Executive

National Instruments Switch Executive is an intelligent switch management and routing application that you can use with TestStand. NI Switch Executive allows you to interactively configure switch devices from multiple vendors as a single virtual device. You can also specify intuitive names for each channel within the virtual switch device and use the end-to-end routing feature to automatically find switch routes by selecting the channels you need to connect.

Use TestStand's IVI Switch step type and the Switching tab on the Step Properties dialog box to automate the defined routes required for each test.

Switching Tab

The Switching tab on the Step Properties dialog box specifies a switching action that TestStand performs around the execution of the step. This feature is only available if you install the National Instruments Switch Executive software. Refer to the *TestStand Help* for more information about the Switching tab on the Step Properties dialog box.

For more information about NI Switch Executive, refer to `ni.com/switchexecutive`.

Route Specification String

When you instruct TestStand to connect or disconnect routes defined in a 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 delimited by ampersands (`&`). National Instruments Switch Executive ignores whitespace characters between tokens in a route specification string.

```
routeOrGroup { & routeOrGroup } { & routeOrGroup } . . .
```

Where `routeOrGroup` is one of the following:

- Route name
- Route group name
- Fully specified path—Enclosed in square brackets and consists of a series of channels delimited by `"->"`. 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 separated by a `"/"` delimiter
- 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 is an example of a route specification string:

```
MyRouteGroup & MyRoute & [Dev1/CH3->CH4,CH4->R0]
```

Step Operations

The IVI Switch step type supports the following IVI switch operations:

- **Connect/Disconnect**—Connects or disconnects the Source and Destination channels in the switch instrument.
- **Configure Scan**—Configures the switch module for scanning.
- **Start Scan**—Initiates a scanning operation.
- **Wait**—Blocks operations until all switches debounce for an instrument.
- **Configure Switch**—Configures channels as Configuration or Source channels, and configures specific paths between channels.
- **Send Software Trigger**—Sends a software trigger command to trigger the instrument during a Scanning operation.

- **Abort Scan**—Cancels a Scanning operation.
- **Get Information**—Retrieves low-level status and information from the instrument.

The IVI Switch step type supports the following Switch Executive operations:

- **Connect/Disconnect**—Connects or disconnects switch routes for a virtual device.
- **Wait**—Blocks until all switches debounce for a virtual device.
- **Get Information**—Retrieves low-level status and information from a virtual device.

Refer to the *TestStand Help* for more information about each of these operations.

Step Properties

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

- **Step.LogicalName**—Contains the logical name expression.
- **Step.InstrOperation**—Contains a value that specifies the operation the step is set to perform.
- **Step.SettingsSource**—Contains the name of the property or variable where the step loads and stores the settings for the operation.
- **Step.IviOperation**—Contains a value that specifies the operation the step is set to perform for IVI Switching mode.
- **Step.ConnectDisconnect**—Contains the settings for the Connect/Disconnect operation.
- **Step.SoftFrontPanel**—Contains the settings for the Show Soft Front Panel operation.
- **Step.GetInfo**—Contains the settings for the Get Information operation.
- **Step.ScanningConfig**—Contains the settings for the Configure Scan operation.
- **Step.Wait**—Contains the settings for the Wait operation.
- **Step.Configure**—Contains the settings for the Configure operation.

IVI Tools

Use the IVI Tools step type to perform low-level operations on an instrument.

Step Operations

The IVI Tools step type supports the following operations:

- **Get Session Info**—Retrieve low-level session references and API class handles to the IVI instrument.
- **Show Soft Front Panel**—Displays the SFP for the tool.
- **Hide Soft Front Panel**—Hides the SFP for the tool.
- **Init**—Initializes the driver or I/O resource for the session.
- **Close**—Closes the IVI session.
- **Reset**—Places the instrument in a known state.
- **Self Test**—Causes the instrument to perform a self-test.
- **Revision Query**—Queries the instrument driver and instrument for revision information.
- **Error Query**—Returns instrument-specific error information.
- **Get Error Info**—Returns error information for the last IVI error for a session.
- **Set/Get/Check Attributes**—Allows you to set, query, or verify the value of attributes.

Refer to the *TestStand Help* for more information about each of these operations.

Step Properties

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

- **Step.LogicalName**—Contains the logical name expression.
- **Step.InstrOperation**—Contains a value that specifies the operation the step is set to perform.
- **Step.SettingsSource**—Contains the name of the property or variable where the step loads and stores the settings for the operation.
- **Step.SoftFrontPanel**—Contains the settings for the Show Soft Front Panel operation. The data type of this property is NI_IviSoftFrontPanel.

- **Step.Init**—Contains the settings for the Init operation.
- **Step.SelfTest**—Contains the settings for the Self Test operation.
- **Step.SessionInfo**—Contains the settings for the Get Session Info operation.
- **Step.RevisionQuery**—Contains the settings for the Get Revision Query operation.
- **Step.ErrorQuery**—Contains the settings for the Error Query operation.
- **Step.ErrorInfo**—Contains the settings for the Get Error Info operation.
- **Step.Attributes**—Contains the settings for the Set/Get/Check Attributes operation.

D

Database Step Types

TestStand includes the following built-in step types that you can use to communicate with a database:

- Open Database
- Open SQL Statement
- Close SQL Statement
- Close Database
- Data Operation
- Property Loader

A simple sequence of Database steps might include the following:

1. Connect to the database using the Open Database step type.
2. Use the Open SQL Statement step type to issue a SQL query on tables in the database.
3. Create new records, then get and update existing records using Data Operation step types.
4. Use the Close SQL Statement step type to close the SQL query.
5. Close the database connection using the Close Database step type.

Note Use the Property Loader step type to import property and variable values from a file or database during an execution.

The following sections describe each Database step type and their custom step properties.

You can view examples of Database step types in the `<TestStand>\Examples\Database` directory. Refer to the *TestStand Help* for more information about each of the Database step types.

Open Database

Use the Open Database step type to open a database for use in TestStand. An Open Database step returns a database handle that you can use to open SQL statements.

Step Properties

The Open Database step type defines the following step properties in addition to the common custom properties:

- **Step.ConnectionString**—Specifies a string expression that contains the name of the data link to open.
- **Step.DatabaseHandle**—Specifies the numeric variable or property assigned to the value of the opened database handle.
- **Step.RequiresParameters**—Specifies whether the SQL statement requires parameters to execute.

Close Database

Use the Close Database step type to close the database handle that you obtain from an Open Database step type.

Tip National Instruments recommends placing Close Database steps in the Cleanup step group.

Step Properties

The Close Database step type defines the following step property in addition to the common custom properties:

- **Step.DatabaseHandle**—Specifies the name of the variable or property that contains the open database handle that is to be closed. The variable or property is of the Number data type.

Note TestStand does not automatically close open database handles. You must call a Close Database step for your open handles. If you abort an execution, you must exit the application process that loaded the TestStand Engine to guarantee that TestStand frees all database handles. Selecting Unload All Modules from the File menu does not close the handles.

Open SQL Statement

After you open a database, you must select a set of data in the database to work with. Use the Open SQL Statement step type to select this data. After you open an Open SQL Statement step, you can perform multiple operations on that data set using the Data Operation steps. An Open SQL Statement step returns a statement handle that you can use in the Data Operation steps.

Step Properties

The Open SQL Statement step type defines the following step properties in addition to the common custom properties:

- **Step.PageSize**—Specifies the number of records in a page for the SQL statement.
- **Step.CommandTimeout**—Specifies the amount of time, in seconds, TestStand waits when attempting to issue a command to the open database connection.
- **Step.CacheSize**—Specifies the cache size for the SQL statement.
- **Step.MaxRecordsToSelect**—Specifies the maximum number of records the SQL statement can return.
- **Step.CursorType**—Specifies the cursor type that the SQL statement uses.
- **Step.CursorLocation**—Specifies where the data source maintains cursors for a connection.
- **Step.MarshalOptions**—Specifies the marshal options for the updated records in the SQL statement.
- **Step.LockType**—Specifies the lock type for the records the SQL statement selects.
- **Step.CommandType**—Specifies the command type of the SQL statement.
- **Step.DatabaseHandle**—Specifies the name of the variable or property that contains the database handle with which you open the SQL statement.
- **Step.StatementHandle**—Specifies the numeric variable or property that is assigned for the value of the SQL statement handle.
- **Step.SQLStatement**—Specifies a string expression that contains a SQL command.
- **Step.NumberOfRecordsSelected**—Specifies the numeric variable or property to which the step assigns the number of records the SQL statement returns.
- **Step.RequiresParameters**—Specifies whether the SQL statement requires input or output parameters to execute. If `False`, the step immediately executes the SQL statement. If `True`, the step only prepares the SQL statement, and a subsequent Data Operation step must perform an Execute operation that defines the parameters for the statement.

Close SQL Statement

Use the Close SQL Statement step to close a SQL statement handle that you obtain from an Open SQL Statement step.

Tip National Instruments recommends placing Close SQL Statement steps in the Cleanup step group.

Step Properties

The Close SQL Statement step type defines the following step property in addition to the common custom properties:

- **Step.StatementHandle**—Specifies the name of the variable or property of type Number that contains the SQL statement handle that is to be closed.

Data Operation

Use the Data Operation step type to perform operations on a SQL statement that you open with an Open SQL Statement step. With the Data Operation step, you can fetch new records, retrieve values from a record, modify existing records, create new records, and delete records. For SQL statements that require parameters, you can create parameters and set input values, execute statements, close statements, and fetch output parameter values.

Custom Properties

The Data Operation step type defines the following step properties in addition to the common custom properties:

- **Step.StatementHandle**—Specifies a string expression that contains the name of the SQL statement to operate on.
- **Step.RecordToOperateOn**—Specifies the record to operate on. Valid values are `0` = New, `1`= Current, `2` = Next, `3` = Previous, `4` = Index.
- **Step.RecordIndex**—Specifies the index of the record to operate on when Step.RecordToOperateOn is set to fetch a specific index.
- **Step.Operation**—Specifies the operation to perform on the record. Valid values are `0` = Fetch only, `1` = Set, `2` = Get, `3` = Put, `4` = Delete, `5` = Set and Put, `6` = Execute, `7` = Close.
- **Step.SQLStatement**—Specifies the SQL statement used by the Edit Data Operation dialog box to populate the ring controls that contain column names.

- **Step.ColumnListSource**—Specifies the name of the variable or property that stores the column-to-variable or column-to-property mappings. The variable or property must be an array of type DatabaseColumnValue. By default, the value is Step.ColumnList.
- **Step.ColumnList**—Specifies the column-to-variable or column-to-property mapping to perform on a Get or Set operation. The property must be an array of type DatabaseColumnValue.

  The DatabaseColumnValue custom data type contains the following subproperties:

  – **ColumnName**—Specifies the name of the column from which to get a value or to which to assign a value.

  – **ColumnNumber**—Specifies the number of the column in the SQL statement.

  – **Data**—Specifies the variable or property to which TestStand assigns the column value or the expression that TestStand evaluates and assigns to the column.

  – **FormatString**—Specifies an optional format string for dates, times, and currencies. Use the empty string (`""`) if you want to use the default format. Refer to the *TestStand Help* for a description of valid format strings.

  – **WriteNull**—Specifies whether to write `NULL` to the column instead of the value in the Data expression property.

  – **Status**—Indicates the error code returned for the Get or Set operation.

  – **Direction**—Contains an enumerated value that specifies whether the parameter direction is `In`, `Out`, `In/Out`, or `Return`.

  – **Type**—Contains an enumerated value that specifies whether the parameter value is `String`, `Number`, `Boolean`, or `Date/Time`.

  – **Size**—Specifies the maximum size of a string parameter.

Note You cannot encapsulate your data operations within a transaction. Transactions are not available in the current version of the TestStand Database step types.

Property Loader

Use the Property Loader step type to dynamically load the values for properties and variables from a text file, a Microsoft Excel file, or a DBMS database at run time. For example, you can develop a common sequence that can test two different models of a cellular phone, where each model requires unique limit values for each step. If you use step properties to hold

the limit values, you can include a Property Loader step in the sequence to load the correct limit values into the step properties.

Place a Property Loader step in the Setup step group of a sequence. This directs the Property Loader step to initialize the property and variable values before the steps in the Main step group execute.

Loading From File

The source of file-based values can be a tab-delimited text file (`.txt`), a comma-delimited text file (`.csv`), or an Excel file (`.xls`). The data is presented in a table format. The following is an example of a tab-delimited limits file with one data block specified by starting and ending data markers.

```
Start Marker

<Step Name>          Limits.Low     Limits.High     Limits.String
Voltage at Pin A     9.0            11.0
Voltage at Pin B     8.5            9.5
Self Test Output                                    "SYSTEM OK"

<Locals>                            Variable Value
Count                               100

<FileGlobals>                       Variable Value
Count                               99

<StationGlobals>                    Variable Value
Power_On                            False

End Marker
```

In the step name section the row names correspond to step names and the column headings correspond to the names of step properties. Not all columns apply to each row, and each row only contains values for the columns that define properties that exist in the row's corresponding step. For variable sections, each row specifies the name of the property and its corresponding value. Starting and ending data markers designate the bounds of the table. A data file can contain more than one block of data.

Use the **Importing/Exporting Properties** command in the **Tools** menu to export property and variable data in the appropriate table format.

Loading From Database

The source of database values is a recordset returned from an Open SQL Statement step. The SQL statement recordset is presented in a table format, where each row pertains to a particular sequence step or to a variable scope, as shown in Table D-1. The column headings correspond to the names of properties in the steps or variable scopes. Not all columns apply to each row. Each row only contains values for the columns that define properties or variables that are actually in the step or variable scope for the row.

Table D-1. Example Data for Property Loader Step

STEPNAME	LIMITS_ HIGH	LIMITS_ LOW	LIMITS_ STRING	POWER_ON	COUNT	SEQUENCE NAME
Voltage at Pin A	9	11	—	—	—	`Phone Test.seq`
Voltage at Pin B	8.5	9.5	—	—	—	`Phone Test.seq`
Self Test Output	—	—	"SYS OK"	—	—	`Phone Test.seq`
<Locals>	—	—	—	—	100	`Phone Test.seq`
<File Globals>	—	—	—	—	99	`Phone Test.seq`
<Station Globals>	—	—	—	False	—	`Phone Test.seq`
Frequency at Pin A	100,000	10,000	—	—	—	`Frequency Test.seq`
Frequency at Pin B	90,000	9,000	—	—	—	`Frequency Test.seq`
Self Test Output	—	—	"OK"	—	—	`Frequency Test.seq`

For database sources, the Property Loader step can filter the data that the SQL statement returns so that you only load values from rows that contain specific column values. This is equivalent to the starting and ending data markers when importing values from a file. For example, in Table D-1 you can load the rows only for rows where the SEQUENCE NAME field contains the value, `Phone Test.seq`.

Custom Properties

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

- **Step.Result.NumPropertiesRead**—Indicates the total number of values that the step loaded from the file or database.
- **Step.Result.NumPropertiesApplied**—Indicates the total number of values the step assigned to properties or variables. A number less than Step.Result.NumPropertiesRead indicates that the step was unable to update properties or variables.
- **Step.ColumnListSource**—Specifies the name of the variable or property that stores the list of column comparisons which you are using to filter the rows in a database recordset. The variable or property must be an array of type DatabaseColumnValue. By default, the value is Step.ColumnList.
- **Step.ColumnList**—Specifies the column comparisons TestStand makes on a recordset before loading its values into properties. This property must be an array of type DatabaseColumnValue.

  The DatabaseColumnValue custom data type contains the following subproperties:

  – **ColumnName**—Specifies the name of the column on which to perform the comparison.

  – **ColumnNumber**—Indicates the number of the column in the recordset.

  – **Data**—Specifies the expression that TestStand evaluates at run time to compare against the column value.

  – **FormatString**—Specifies an optional format string for dates, times, and currencies. Use an empty string (`""`) if you want to use the default format. Refer to the *TestStand Help* for a description of valid format strings.

  – **Direction**—Contains an enumerated value that specifies whether the parameter direction is `In`, `Out`, `In/Out`, or `Return`.

  – **Type**—Contains an enumerated value that specifies whether the parameter type is `String`, `Number`, `Boolean`, or `Date/Time`.

  – **Size**—Specifies the maximum size of a string parameter.

  – **WriteNull**—Not used.

  – **Status**—Not used.

- **Step.PropertiesListSource**—Specifies the name of the variable or property that stores the list of variables and properties into which to load data. The variable or property must be an array of type DatabasePropertyMapping. By default, the value is Step.PropertiesList.
- **Step.PropertiesList**—Specifies the list of variables and properties in which to load data. The list must be an array of type DatabasePropertyMapping. Each element of the array defines a mapping between the source data and a TestStand variable or property.

  The DatabasePropertyMapping custom data type contains the following subproperties:

  – **PropertyName**—Specifies the name of the property or variable to which a value is assigned.

  – **PropertyType**—Specifies the scope of the property or variable, such as step, local, file global, or station global. Valid values are: `0` = Step, `1` = Local, `2` = File Global; `3` = Station Global.

  – **DataType**—Specifies the TestStand type of the property. Valid values are: `1` = String, `2` = Boolean, `3` = Number.

  – **ColumnName**—Specifies the name of the column from which the value is obtained.
- **Step.Database.SQLStatementHandle**—Specifies the name of the variable or property that contains the SQL statement handle the step uses at run time to load values.
- **Step.Database.SQLStatement**—Specifies the SQL statement the Edit Property Loader dialog box uses to populate ring controls that contain column names.
- **Step.Database.StepNameColumn**—Specifies the name of the column in the recordset that contains the names of the steps and variable scopes that define the rows of data.
- **Step.Database.AppendTypeName**—Specifies whether TestStand appends the data type name of the property to the column name when selecting a property from the available list.
- **Step.Database.MaxColumnSize**—Specifies the maximum number of characters for a column name.
- **Step.Database.FilterUsingColumnList**—Specifies if the step only loads the rows that match the specific column value.
- **Step.File.Path**—Specifies a literal pathname for the data file.
- **Step.File.DecimalPoint**—Specifies the type of decimal point the file uses.

- **Step.File.UseExpr**—Specifies whether to use Step.File.Path or Step.File.FileExpr for the pathname of the data file.
- **Step.File.FileExpr**—Specifies a pathname expression that TestStand evaluates at run time.
- **Step.File.Format**—Specifies the type of delimiters in the file and the file type. The possible values are `Tab`, `Comma`, or `Excel`.
- **Step.File.Start.MarkerExpr**—Specifies the expression for the starting marker.
- **Step.File.EndMarkerExpr**—Specifies the expression for the ending marker.
- **Step.File.Skip**—Specifies the string that, when it appears at the beginning of a row, causes the step type to ignore the row.
- **Step.File.MapColumnsUsingFirstRow**—Specifies whether the first row of each data block in the data file contains the names of the step properties into which the step loads the property values.
- **Step.File.ColumnMapping**—Specifies the names of the properties into which the step loads the values if Step.File.MapColumnsUsingFirstRow is `False`.

E

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources include the following:
  - **Self-Help Resources**—For immediate answers and solutions, visit our extensive library of technical support resources available in English, Japanese, and Spanish at ni.com/support. These resources are available for most products at no cost to registered users and include software drivers and updates, a KnowledgeBase, product manuals, step-by-step troubleshooting wizards, conformity documentation, example code, tutorials and application notes, instrument drivers, discussion forums, a measurement glossary, and so on.
  - **Assisted Support Options**—Contact NI engineers and other measurement and automation professionals by visiting ni.com/support. Our online system helps you define your question and connects you to the experts by phone, discussion forum, or email.
- **Training**—Visit ni.com/training for self-paced tutorials, videos, and interactive CDs. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, NI Alliance Program members can help. To learn more, call your local NI office or visit ni.com/alliance.

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

Glossary

A

abort	To stop an execution without running any of the steps in the Cleanup step groups in the sequences on the call stack. When you abort an execution, no report generation occurs.
active window	The window that user input affects at a given moment. The title of an active window is highlighted.
ActiveX (Microsoft ActiveX)	Set of Microsoft technologies for reusable software components. Formerly called *OLE*.
ActiveX control	A reusable software component that adds functionality to any compatible ActiveX control container.
ActiveX/COM Adapter	*See* adapter.
ActiveX reference property	A container of information that maintains a reference to an ActiveX object. TestStand maintains the value of the property as an `IDispatch` or `IUnknown` pointer.
ActiveX server	Any executable code that makes itself available to other applications according to the ActiveX standard. ActiveX implies a client/server relationship in which the client requests objects from the server and asks the server to perform actions on the objects.
adapter	If an adapter is specific to an application development environment (ADE), the adapter knows how to open the ADE, how to create source code for a new code module in the ADE, and how to display the source for an existing code module in the ADE. Some adapters support stepping into the source code of the ADE while executing the step from the TestStand Sequence Editor.
administrator	A user profile that usually contains all privileges for a test station.
Application Development Environment (ADE)	A programming environment such as LabVIEW, LabWindows/CVI, or Microsoft Visual Basic, in which you can create code modules and operator interfaces.

ADO	ActiveX Data Object.
Application Programming Interface (API)	A set of classes, methods, and properties that you use to control a specific service, such as the TestStand Engine.
array property	A property that contains an array of single-valued properties of the same type.
ASCII	American Standard Code for Information Interchange.

B

binding	*See* early binding and late binding.
block diagram	Pictorial description or representation of a program or algorithm. In LabVIEW, the block diagram, which consists of executable icons called nodes and wires that carry data between the nodes, is the source code for the VI. The block diagram resides in the Diagram window of the VI.
breakpoint	An interruption in the execution of a program.
built-in property	A property that all steps or sequences contain. For example, the step run mode property. TestStand normally does not display these properties in the sequence editor, although it lets you modify some of them through dialog boxes.
built-in step type property	A property that is common to all steps of the same type. A built-in step type property is either a class step type property or an instance step type property.
button	A dialog box item that, when selected, executes a command associated with the dialog box.

C

C/C++ DLL Adapter	*See* adapter.
call stack	The chain of active sequences that are waiting for the nested subsequences to complete.
Call Stack pane	Displays the call stack for the execution thread that is currently selected in the Threads control.

checkbox	An input control in a dialog box that allows you to toggle between two possible options.
class	Defines a list of methods and properties that you can use with respect to the objects that you create as instances of that class. A class is like a data type definition except that it applies to objects rather than variables.
class step type property	A built-in step property that only exists in the step type itself. TestStand uses these properties to define how the step type works for all step instances. Step instances do not contain their own copies of class properties.
client sequence file	A sequence file that contains the Main sequence that a process model invokes to test a UUT. Each client sequence file contains a sequence called `MainSequence`. The process model defines what is constant about your testing process, whereas the client sequence file defines the steps that are unique to the different types of tests you run.
clipboard	A temporary storage area the operating system uses to hold data that you cut, copy, or delete from a work area.
cluster	A set of ordered, non-indexed data elements in LabVIEW of any data type including numeric, Boolean, string, array, or cluster. The elements must be all controls or all indicators.
code module	A program module, such as a Windows Dynamic Link Library (`.dll`) or LabVIEW VI (`.vi`), that contains one or more functions that perform a specific test or other action.
code template	A source file that contains skeleton code. The skeleton code serves as a starting point for the development of code modules for steps that use a particular step type.
Configuration entry point	A sequence in the process model file that configures a feature of the process model. Configuration entry points usually save configuration information in a `.ini` file in the `<TestStand>\Cfg` directory. By default, Configuration entry points are listed in the Configure menu. For example, the default process model contains the Configuration entry point Config Report Options. The Config Report Options entry point is listed as Report Options in the Configure menu.
connection string	A string version of the connection information required to open a session to a database.

connector	Part of a LabVIEW VI or function node that contains its input and output terminals, through which data passes to and from the node.
container property	A property that contains no values, and typically contains multiple subproperties. Container properties are analogous to structures in Microsoft Visual C/C++ and to clusters in LabVIEW.
context menu	Access context menus by right-clicking on an object. Menu options in a context menu pertain specifically to the object you have selected.
control	An input and output device in a panel or window, in which you can enter data or make a setting.
control flow	The sequential order of instructions that determines execution order.
custom named data type	A data type that you define and name. For example, you might create a `Transmitter` data type that contains subproperties such as number of channels, `NumChannels`, and power level, `PowerLevel`.
custom property	A property that you define in a step type. Each step you create with the step type has its own copy of the custom property. TestStand uses the value that you specify for the custom property in the step type as the initial value of the property in each new step you create. Normally, after you create a step, you can change the value of the property in the step.

D

data link	Connection information for opening a data source, such as a database. A data link typically specifies the server on which the data resides, the database or file that contains the data, the user ID, and permissions to use when connecting to the data source.
data source	A provider of data, such as a database.
DBMS	Database Management Systems.
developer	A user profile that usually contains all privileges associated with operating, debugging, and developing sequences and sequence files, but excludes configuration of user privileges, report options, and database options.

dialog box	A user interface in which you specify additional information for the completion of a command.
DLL	Dynamic Link Library

E

early binding	Setting that causes the ActiveX/COM Adapter to use IDs to specify to automation servers what operations to perform on objects. *See* late binding.
Edit substep	A substep that the engine calls when a developer or user edits the step. You invoke the substep with the menu item that is listed in the context menu above Specify Module. The Edit substep displays a dialog box in which the sequence developer edits the values of custom step properties. For example, the Edit Limits item is listed in the context menu for Numeric Limit Test steps, and the Edit Pass/Fail Source item is listed in the context menu for Pass/Fail Test steps.
engine	A module or set of modules that provide an API for creating, editing, executing, and debugging sequences. A sequence editor or operator interface uses the services of a test executive engine.
Engine callback	A sequence that TestStand invokes at specific points during execution. Use Engine callbacks to tell TestStand to call certain sequences before and after the execution of individual steps, before and after interactive executions, after loading a sequence file, and before unloading a sequence file.
entry point	A sequence in the process model file that TestStand displays as a menu item, such as Test UUTs, Single Pass, and Report Options.
error occurred flag	A Boolean flag, Step.Result.Error.Occurred, that indicates whether a run-time error occurred in a step.
execution	An object that contains information TestStand needs to run a sequence, its steps, and any of the sequences it calls. You can suspend, interactively debug, resume, terminate, or abort executions.
Execution entry point	A sequence in a process model that runs tests against a UUT. Execution entry points call the MainSequence callback in the client sequence file. The default process model contains two execution entry points: Test UUTs and Single Pass. By default, Execution entry points are listed in the Execute menu. Execution entry points are only visible in the menu when the active window contains a sequence file that has a MainSequence callback.

Execution object	An object that contains all of the information TestStand needs to run a sequence, its steps, and any subsequences it calls. Typically, the TestStand Sequence Editor creates a new window for each execution.
Execution window	A window in the sequence editor or operator interface that displays the steps an execution runs. When execution is suspended, the Execution window displays the next step to execute and provides single-stepping options. You can also view variables and properties in any active sequence context in the call stack.
expression	A formula that calculates a new value from the values of multiple variables or properties. In expressions, you can access all of the variables and properties in the sequence context that is active when TestStand evaluates the expression.

F

Front-End callback	A common sequence that the sequence editor and operator interfaces call. Front-End callbacks allow multiple applications to share the same implementation for a specific operation. TestStand installs the sequence file `FrontEndCallback.seq`, which contains the Front-End callback sequence, LoginLogout.
Front-End callback sequence file	A sequence file that contains Front-End callbacks. TestStand installs the sequence file `FrontEndCallback.seq`, which contains the Front-End callback sequence, LoginLogout.
front panel	The interactive user interface of a LabVIEW VI. Modeled from the front panel of physical instruments, it is composed of switches, slides, meters, graphs, charts, gauges, LEDs, and other controls and indicators.

G

global variable	TestStand defines two types of global variables: sequence file globals and station globals. Sequence file globals are accessible by any sequence or step in the sequence file. Station globals are accessible by any sequence file loaded on the station. The values of station global variables are persistent across different executions and even across different invocations of TestStand.
GUI	*See* operator interface.

H

hex	hexadecimal.
hidden Execution entry point	Execution entry points used by the main Execution entry points to initiate test socket executions. These entry points are never displayed.
highlight	The way in which input focus appears on a TestStand screen. To move the input focus onto an item.
HTBasic Adapter	*See* adapter.

I

`IDispatch` pointer	A pointer to an interface that exposes objects, methods, and properties to automation programming tools and other applications.
in-process	When executable code runs in the same process space as the client, in other words, an ActiveX server in a dynamic link library (DLL).
instance step type property	A built-in step property that exists in each step instance. Each step that you create with the step type has its own copy of the property. TestStand uses the value you specify for an instance step type property in the step type as the initial value of the property in each new step that you create. Normally, after you create a step, you can change the values of its instance step type properties.
interactive mode	When you run steps by selecting one or more steps in a sequence and select Run Selected Steps or Loop Selected Steps from the context menu or menu bar. The selected steps in the sequence execute, regardless of any branching logic that the sequence contains. The selected steps run in the order in which they appear in the sequence.
`IUnknown` pointer	An interface, provided by all ActiveX objects, that enables you to control the lifetime and obtain other interfaces of an object.

K

kill	To stop a running, terminating, or aborting execution by terminating the thread of the execution without any cleanup of memory. This action can leave TestStand in an unreliable state.

L

LabVIEW	Laboratory Virtual Instrument Engineering Workbench. A program development application based on the G programming language and used commonly for test and measurement purposes.
LabVIEW Adapter	*See* adapter.
LabWindows/CVI Adapter	*See* adapter.
late binding	Setting that causes the ActiveX/COM Adapter to use names to specify to a server what operations to perform on objects. *See* early binding.
legacy	Characteristic of a version of TestStand prior to TestStand 3.0.
list box	A control that displays a list of possible choices.
local variable	A property of a sequence that holds a value or additional subproperties. Only a step within the sequence can directly access the property value.

M

Main sequence	The sequence that initiates the tests on a UUT. The process model invokes the Main sequence as part of the overall testing process. The process model defines what is constant about your testing process, whereas the Main sequence defines the steps that are unique to the different types of tests you run.
manager controls	Controls that call the TestStand API to perform tasks such as loading files, launching executions, and retrieving sequence information. Manager controls are visible at design time but invisible at run time.
MB	Megabytes of memory.
menu bar	Horizontal bar that contains names of main menus.
method	Performs an operation or function on an object.
MFC	Microsoft Foundation Class Library.
Model callback	A mechanism that allows a sequence file to customize the default behavior of a sequence in the process model.

Model sequence file	A special type of sequence file that contains process model sequences. The sequences within the Model sequence file direct the high-level sequence flow of an execution when you test a UUT.
Module adapter	A module adapter knows how to load and call a code module, how to pass parameters to a code module, and how to return values and status from a code module.
multi-threaded apartment model (MTA)	A model in which an ActiveX object can be accessed from any thread at any time. The operating system does not synchronize access to the object. You cannot create or display a window that contains an ActiveX control from a thread that is initialized using the mutli-threaded model. ActiveX controls require threads initialized using the single-threaded apartment model. By default, all execution threads are initialized using the multi-threaded apartment model.

N

named data type	A type of variable or property that you give a unique name. The data type usually contains multiple subproperties, thus creating an arbitrarily complex data structure. All variables or properties that use the data type have the same data structure, but the values they contain can differ.
nested	Called by another step or sequence. If a sequence calls a subsequence, the subsequence is nested in the invocation of the calling sequence.
nested interactive execution	When you run steps interactively from an Execution window for a normal execution that is suspended at a breakpoint. You can run steps only in the sequence and step group in which execution is suspended. The selected steps run within the context of the normal execution.
.NET Adapter	*See* adapter.
normal execution	When you start an execution in the sequence editor by selecting Run <Sequence Name>, where <Sequence Name> is the name of the sequence that you are running, or one of the process model entry points from the Execute menu.
normal sequence file	Any sequence file containing sequences that test UUTs.
numeric property	A 64-bit, floating-point value in the IEEE 754 format.

O

object	A service that an ActiveX server makes available to clients.
ODBC	Open Database Connectivity.
OLE-DB	Object Linking and Embedding Database.
operator	A user profile that usually contains all privileges associated with operating a test station, but excludes debugging of sequence executions, editing of sequence files, and configuration of user privileges, station options, report options, and database options.
operator interface	A program that provides a graphical user interface (GUI) for executing sequences on a production station.
out-of-process	When executable code does not run in the same process space as the client, such as an ActiveX server in an executable.

P

pop-up menus	*See* context menu.
post actions	Actions that TestStand takes depending on the pass/fail status of the step or a custom condition that the engine evaluates after executing a step. Post actions allow you to execute callbacks or jump to other steps after executing the step.
Post-Step substep	A substep that the engine invokes after calling a code module. For example, a Post-Step substep might call a code module that compares the values the code module stored in step properties against limit values that the Edit substep stored in other step properties.
Pre-Step substep	A substep that the engine invokes before calling the code module. For example, a Pre-Step substep might call a code module that retrieves measurement configuration parameters and stores them into step properties for use by the code module.
preconditions	A set of conditions for a step that must be `True` for TestStand to execute the step during the normal flow of execution in a sequence.

process — A running application that consists of a private memory space and other operating-system resources that are visible to the process. A process also contains one or more threads that run in the context of the process.

process model — A sequence file you designate that performs a standard series of operations before and after a test executive executes the sequence that performs the tests. Common operations include identifying the UUT, notifying the operator of pass/fail status, generating a test report, and logging results.

property — A container of information, which stores and maintains a setting or attribute of an object. A property can be of type number, string, Boolean, container, ActiveX reference, a user-defined data type, or an array of these types. A property can contain a single value, an array of values of the same type, or no value at all. A property can also contain any number of subproperties. Only a container property has the ability to contain any number of subproperties. Each property has a name and a comment.

property-array property — A property containing a value that is an array of subproperties of a single type. In addition to the array of subproperties, property-array properties can contain any number of subproperties of other types.

R

record set — The retrieved records that an SQL SELECT command or query returns.

reference count — Information that each ActiveX object uses to keep track of the number of things that reference it. This allows the object to determine when to free the resources it uses.

reference property — *See* ActiveX reference property.

resource string — Text strings stored in an external file so that you can alter the strings without directly altering the application.

root interactive execution — When you run selected steps from a Sequence File window in an independent execution. Root interactive executions do not invoke process models.

RTF — Rich Text Format.

run mode — The mode in which you execute a step, such as normal, skip, force pass, or force fail.

run-time error	An error condition that forces an execution to terminate. When the error occurs while running a sequence, TestStand jumps to the Cleanup step group, and the error propagates to any calling sequence up the call stack to the top-level sequence.
RunState	Contains properties that describe the state of execution in the sequence invocation.

S

s	seconds.
sequence	Located within a sequence file, a sequence contains a series of steps that you specify to execute in a particular order. When and if a step is executed can depend on the results of previous steps.
Sequence Adapter	*See* adapter.
sequence context	A TestStand object that contains references to all global variables and all local variables and step properties in active sequences. The contents of the sequence context changes depending on the currently executing sequence and step.
sequence editor	A program that provides a graphical user interface (GUI) for creating, editing, and debugging sequences.
sequence file	A file that contains the definition of one or more sequences.
Sequence File window	A separate window within the sequence editor in which a sequence file appears.
single-threaded apartment model (STA)	A model in which ActiveX objects execute within a single thread. The operating system synchronizes all access to the object when accessing the object from other threads. You must create objects that use the single-threaded apartment model, such as ActiveX controls, in threads that are initialized to use the single-threaded apartment model. TestStand executes Edit substeps in threads initialized to use the single-threaded apartment model to allow the substep to display windows that include ActiveX controls.
single-valued property	A property that contains a single value. TestStand has four types of these properties: number, string, Boolean, and object reference.
soft front panel (SFP)	A software graphical display for an instrument.

source code template	A set of source files that contain skeleton code, which serves as a starting point for the development of code modules for steps. TestStand uses the source code template when you click Create Code on the Source Code tab on the Specify Module dialog box for a step.
SQL Null	An empty column in a row in a database table.
SQL Statement data	The retrieved records that an SQL SELECT command or query returns.
standard named data type	A data type that TestStand defines and names. You can add subproperties to the standard data types, but you cannot delete any of their built-in subproperties. The standard named data types are `Path`, `Error`, and `CommonResults`.
station	A complete TestStand test implementation that operators, developers, and administrators use to perform tests.
Station callback sequence file	A sequence file that contains the Station callback sequences. Station callbacks run before and after the engine executes each step in any normal or interactive execution.
station global variables	Variables that are persistent across different executions and even across different invocations of the sequence editor or operator interfaces. The TestStand Engine maintains the value of station global variables in a file on the run-time computer.
station model	A process model that you select to use for all sequence files for a station. The TestStand installation program establishes `SequentialModel.seq` as the default station model file. Use the Station Options dialog box to select a different station model.
step	An element that you can insert into a sequence that performs an action, such as calling a code module to perform a specific test. Typically, a sequence contains a series of steps that define your test and execution flow.
step group	A set of steps in a sequence. A sequence contains the following groups of steps: Setup, Main, and Cleanup. When TestStand executes a sequence, the steps in the Setup step group execute first, the steps in the Main step group execute next, and the steps in the Cleanup step group execute last.
step property	A property of a step.

step result	A container property that contains a copy of the subproperties from the Result property of a step and additional execution information such as the name of the step and its position in the sequence. TestStand automatically creates a step result as each step executes and places the step result into a result list that TestStand uses to generate its reports.
step status	A string value that indicates the status of a step in an execution. Every step in TestStand has a Result.Status property. Although TestStand does not impose restrictions on the values to which the step or its code module can set the status property, TestStand and the built-in step types use and recognize a predefined set of values.
step type	A component that defines a set of custom step properties and standard behavior for each step of that type. All steps of the same type have the same properties, but the values of the properties can differ. Step types define their standard behaviors using substeps.
step-type-specific dialog box	A dialog box that step types display when you invoke their Edit substep. The dialog box lets you modify step properties that are specific to the step type. You invoke the dialog box with the menu item that is listed in the context menu above Specify Module. For example, the Edit Limits item is listed in the context menu for Numeric Limit Test steps, and the Edit Pass/Fail Source item is listed in the context menu for Pass/Fail Test steps.
subsequence	A sequence that another sequence calls. You specify a subsequence call as a step in the calling sequence.
substeps	Actions that a step type performs for a step other than calling the code module. You define a substep by selecting an adapter and specifying a module call. TestStand defines three different types of substeps: Edit substep, Pre-Step substep, and Post-Step substep.
substep module	The code module that a Edit, Pre-Step, or Post-Step substep calls.

T

technician	A user profile that usually contains all privileges associated with operating and debugging sequences and sequence files, but excludes editing of sequence files and configuration of user privileges, station options, report options, and database options.
template	*See* code template.
terminal	Object or region on a LabVIEW VI node through which data passes.
terminate	To stop an execution by halting the normal execution flow and running all the Cleanup step groups in the sequences on the call stack.
test executive engine	*See* engine.
TestStand UI Controls Library	Provides a set of common commands that you can add to your application. Connect these commands to a TestStand button or application menu item to automatically execute the command.
ThisContext	Holds a reference to the current sequence context. You usually use this property to pass the entire sequence context as an argument to a subsequence or a code module. *See also* sequence context.
thread	A subprocess that is part of a process or application. A thread can execute any part of the code of an application, and other threads within a process execute concurrently. All threads under a process share the memory space and other operating-system resources of their respective processes.

U

Unit Under Test (UUT)	The device or component that you are testing.
User Manager	The component of the TestStand Engine that maintains a list of users, their login names and passwords, and their privileges. You can access the user manager from the User Manager window in the sequence editor.

V

variables	A property that you can freely create in a certain context. You can have variables that are global to a sequence file or local to a particular sequence. You can also have station global variables.
variables window	A window that shows the values of all the currently active variables or properties.
variant	Data type that can hold any defined type of data.
VI	Virtual Instrument.
VI library	Special file of type `.LLB` that contains a collection of related VIs for a specific use.
visible controls	Controls that connect to manager controls to automatically display information or to enable the user to select items to view.

W

Watch Expression pane	A pane that shows the values of user-selectable variables and expressions that are currently active.
window	A working area that supports specific tasks related to developing and executing programs.
wire	Tool used in LabVIEW to define data paths between source and sink terminals.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X