NATIONAL INSTRUMENTS™
TestStand™

Getting Started with TestStand

NATIONAL INSTRUMENTS™

March 2001 Edition
Part Number 322017B-01

Worldwide Technical Support and Product Information

`ni.com`

National Instruments Corporate Headquarters

11500 North Mopac Expressway   Austin, Texas 78759-3504   USA   Tel: 512 794 0100

Worldwide Offices

Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 284 5011, Canada (Calgary) 403 274 9391, Canada (Ottawa) 613 233 5949, Canada (Québec) 514 694 8521, Canada (Toronto) 905 785 0085, China (Shanghai) 021 6555 7838, China (ShenZhen) 0755 3904939, Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24, Germany 089 741 31 30, Greece 30 1 42 96 427, Hong Kong 2645 3186, India 91805275406, Israel 03 6120092, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 280 7625, Netherlands 0348 433466, New Zealand 09 914 0488, Norway 32 27 73 00, Poland 0 22 528 94 06, Portugal 351 1 726 9011, Singapore 2265886, Spain 91 640 0085, Sweden 08 587 895 00, Switzerland 056 200 51 51, Taiwan 02 2528 7227, United Kingdom 01635 523545

For further support information, see the *Technical Support Resources* appendix. To comment on the documentation, send e-mail to `techpubs@ni.com`

Important Information

Conventions

The following conventions are used in this manual:

»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.
✎	This icon denotes a note, which alerts you to important information.
⚠	This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.
bold	Bold text denotes items that you must select or click on 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 bold`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.
`monospace italic`	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

Contents

Chapter 1
Introduction to TestStand

Installing TestStand 1-1
- Minimum System Requirements 1-1
- Installing TestStand 1-1
- What the Setup Programs Install 1-2

Learning TestStand 1-3
TestStand System Overview 1-4
Major Software Components of TestStand 1-5
- TestStand Sequence Editor 1-6
- TestStand Run-Time Operator Interfaces 1-6
- TestStand Test Executive Engine 1-7
- Module Adapters 1-7
- Process Models 1-8

Chapter 2
Loading and Running Sequences

Starting TestStand 2-1
- Introduction to the Sequence Editor 2-2
  - Menu Bar 2-2
  - Toolbar 2-3
- Development Workspace 2-3
- Status Bar 2-3

Loading a Sequence File 2-4
- About Sequences 2-7

Running a Sequence 2-7
- Setting Up Tracing Options 2-7
- Running a Sequence Directly 2-9
- Running a Sequence Using the Sequential Process Model 2-11
- Running a Sequence Using the Batch Process Model 2-12

Chapter 3
Editing Steps in a Sequence

Setting Up the Example 3-1
Adding a New Step 3-1
Specifying the Test Module 3-4
Changing Step Properties 3-5
Calling a Subsequence from a Sequence 3-16

Chapter 4
Debugging Sequences

Setting Up the Example 4-1
Step Mode Execution 4-1

Chapter 5
Using Variables and Properties

Setting Up the Example 5-1
Using TestStand Variables 5-1
Using the Context Tab 5-8
Using the Watch Expression Pane 5-11

Chapter 6
Creating and Debugging Tests

Debugging a LabVIEW VI Using the LabVIEW Standard Prototype Adapter 6-1
    Setting Up the Example 6-2
    Creating a Virtual Instrument Code Module 6-2
    Debugging a Virtual Instrument Code Module 6-11
Debugging a LabVIEW DLL Function Using the DLL Flexible Prototype Adapter 6-12
    Creating the Virtual Instrument Code 6-13
    Building a LabVIEW DLL Code Module 6-16
    Calling the LabVIEW DLL function 6-18
    Debugging the DLL Function 6-23
Debugging a LabWindows/CVI DLL Using the C/CVI Standard Prototype Adapter 6-23
    Setting Up the Example 6-24
    Creating a C/CVI Code Module Test 6-24
    Debugging a CVI Code Module 6-35
Debugging a LabWindows/CVI DLL Using the DLL Flexible Prototype Adapter 6-37
    Setting Up the Example 6-37
    Creating the LabWindows/CVI Code Module 6-37
    Building a LabWindows/CVI DLL 6-44
    Debugging the DLL Function 6-48

Chapter 7
Using Run-Time Operator Interfaces

Loading Sequences 7-1
Running and Debugging Sequences 7-4
Running Multiple Executions 7-6

Chapter 8
Using Callbacks

Setting Up the Example 8-1
Overriding a Process Model Callback 8-1

Chapter 9
Adding Users and Setting Privileges

Setting Up the Example 9-1
Using the User Manager 9-1

Chapter 10
Using ActiveX in Code Modules

Using ActiveX in LabVIEW Test Virtual Instruments 10-2
    Setting Up the Example 10-2
    Creating the Sequence and Virtual Instrument Tests 10-2
    Running the Sequence 10-12
Using ActiveX in LabWindows/CVI Code Modules 10-14
    Setting Up the Example 10-14
    Creating the Sequence and Tests 10-14
    Running the Sequence 10-21

Chapter 11
Additional Development Features

Setting Up the Example 11-1
Interactive Execution 11-1
    Running Selected Steps as a Separate Execution 11-1
    Running Selected Steps During an Execution 11-4
Calling Sequences Dynamically and Passing Parameters 11-6
    Adding a Step to Sequence 11-6
    Running a Sequence 11-10

Chapter 12
Customizing the Report

Setting Up the Example 12-1
Configuring Test Report Options 12-1
Using External Report Viewers 12-6
Adding New Step Properties to a Report 12-7
    Setting Up the Example 12-7
    Creating a Step Type 12-7

Creating a Step Module Using the LabVIEW Standard Prototype Adapter ... 12-10
Creating a Step Module Using the C/CVI Standard Prototype Adapter ... 12-13
Creating a Step Module Using the DLL Flexible Prototype Adapter ... 12-16
Adding to a Report Using Callbacks ... 12-21

Chapter 13
Converting LabVIEW and LabWindows/CVI Test Executive Sequences
Converting LabVIEW Test Executive Sequences ... 13-1
Converting LabWindows/CVI Test Executive Sequences ... 13-2

Appendix A
Technical Support Resources

Glossary

Index

Figures

Figure 1-1. TestStand System Architecture ... 1-5

Figure 2-1. Login Dialog Box ... 2-1
Figure 2-2. Sequence Editor Main Window ... 2-2
Figure 2-3. Sequence Editor Toolbar ... 2-3
Figure 2-4. Sequence Editor Status Bar ... 2-3
Figure 2-5. Open Dialog Box ... 2-4
Figure 2-6. Sample1.seq Sequence File Window ... 2-5
Figure 2-7. Sequence File View Ring ... 2-6
Figure 2-8. Execution Tab on the Station Options Dialog Box ... 2-8
Figure 2-9. Sample1.seq Execution Window ... 2-9
Figure 2-10. Test Simulator Dialog Box ... 2-10
Figure 2-11. Report Generating Status Bar ... 2-11
Figure 2-12. Model Options Dialog Box ... 2-13

Figure 3-1. Selecting the Module Adapter ... 3-2
Figure 3-2. Inserting a New Step ... 3-3
Figure 3-3. Specify Module Dialog Box ... 3-5
Figure 3-4. Step Properties Dialog Box ... 3-6
Figure 3-5. Preconditions Dialog Box ... 3-7
Figure 3-6. Run Options Tab ... 3-8
Figure 3-7. Post Actions Tab ... 3-11

Figure 3-8. Loop Options Tab....................3-12
Figure 3-9. Synchronization Tab....................3-14
Figure 3-10. Expressions Tab....................3-15
Figure 3-11. Edit Sequence Call Dialog Box....................3-17

Figure 4-1. Configure Message Box Step Dialog Box....................4-3
Figure 4-2. Paused Execution of Sample3.seq....................4-4
Figure 4-3. Single-Stepping Toolbar Buttons....................4-5
Figure 4-4. Steps View while Suspended in Subsequence....................4-6

Figure 5-1. Insert Local Context Menu Command....................5-2
Figure 5-2. Edit Statement Step Dialog Box....................5-3
Figure 5-3. Expression Browser Dialog Box....................5-4
Figure 5-4. Preconditions for the Loop End Step....................5-7
Figure 5-5. Context Tab....................5-9
Figure 5-6. Updated Watch Window Pane....................5-11

Figure 6-1. Adapter Configuration....................6-2
Figure 6-2. LabVIEW Adapter Configuration....................6-3
Figure 6-3. LabVIEW Step Module Information....................6-4
Figure 6-4. New Clock Frequency VI in LabVIEW....................6-5
Figure 6-5. Completed Clock Frequency.vi Front Panel....................6-7
Figure 6-6. Clock Frequency.vi Block Diagram....................6-8
Figure 6-7. Edit Numeric Limits Test Dialog Box....................6-9
Figure 6-8. Numeric Format Dialog Box....................6-10
Figure 6-9. LabVIEW Highlight Execution Mode....................6-11
Figure 6-10. Clock Frequency Function Front Panel....................6-14
Figure 6-11. Clock Frequency Function Diagram....................6-15
Figure 6-12. VI Properties Dialog Box....................6-16
Figure 6-13. Define VI Prototype Dialog Box....................6-17
Figure 6-14. Edit DLL Call Dialog Box....................6-19
Figure 6-15. Edit Numeric Limits Test Dialog Box....................6-21
Figure 6-16. Numeric Format Dialog Box....................6-22
Figure 6-17. Adapter Configuration....................6-24
Figure 6-18. C/CVI Standard Adapter Configuration....................6-25
Figure 6-19. Edit CVI Module Call—Module Tab....................6-26
Figure 6-20. Edit CVI Module Call—Source Code Tab....................6-27
Figure 6-21. Generated Result from Create Code Command....................6-28
Figure 6-22. Edit Numeric Limits Test Dialog Box....................6-33
Figure 6-23. Numeric Format Dialog Box....................6-34
Figure 6-24. Stepping into the GetFrequency Function....................6-36
Figure 6-25. Edit DLL Dialog Box for a LabWindows/CVI Code Module....................6-38
Figure 6-26. Choose Code Template Dialog Box....................6-41

Figure 6-27. Prototypes Conflict Dialog Box 6-42
Figure 6-28. Edit Numeric Limits Test Dialog Box 6-43
Figure 6-29. Numeric Format Dialog Box 6-44
Figure 6-30. Debugging the ClockFrequency function 6-49

Figure 7-1. LabWindows/CVI Operator Interface 7-2
Figure 7-2. Open Sequence in Operator Interface 7-3
Figure 7-3. Paused Execution in Operator Interface 7-5

Figure 8-1. TestStand Sequential Model Callbacks 8-3
Figure 8-2. Test UUTs Sequence 8-4
Figure 8-3. Adding Callbacks to a Sequence 8-6

Figure 9-1. User Manager Window 9-3
Figure 9-2. New User Dialog Box 9-4
Figure 9-3. Configure Privileges in New Profile 9-5

Figure 10-1. Insert Locals Array of Numeric 10-3
Figure 10-2. Array Bounds Dialog Box 10-3
Figure 10-3. GenerateWaveform.vi Front Panel 10-5
Figure 10-4. TestStand Control Palette 10-6
Figure 10-5. GenerateTestStandWaveform.vi Control Panel 10-7
Figure 10-6. GenerateWaveform.vi Block Diagram 10-8
Figure 10-7. TestStand Function Palette 10-9
Figure 10-8. GenerateTestStandWaveform.vi Block Diagram 10-10
Figure 10-9. DisplayTestStandWaveform.vi Front Panel 10-11
Figure 10-10. DisplayTestStandWaveform.vi Block Diagram 10-12
Figure 10-11. Locals.Arraydata 10-13
Figure 10-12. Insert Locals Array of Numeric 10-15
Figure 10-13. Array Bounds Dialog Box 10-15
Figure 10-14. Generated GenerateTestStandWaveform Source 10-17
Figure 10-15. Locals.Arraydata Values 10-22

Figure 11-1. Selecting Multiple Steps in a Sequence File Window 11-2
Figure 11-2. Breakpoint During Interactive Execution 11-3
Figure 11-3. Loop on Selected Steps During Execution 11-4
Figure 11-4. Dynamically Calling with an Expression 11-9
Figure 11-5. Dynamically Calling a Sequence 11-10
Figure 11-6. INTELProcessor.seq in the Call Stack Pane 11-11
Figure 11-7. Sequence Parameters in the Context Tab 11-12

Figure 12-1. UUT Report Setting 12-2
Figure 12-2. Test Report in Text Format 12-4

Figure 12-3. Test Report in HTML Format 12-5
Figure 12-4. Insert Numeric Array Context Menu 12-8
Figure 12-5. Numeric Array Properties Context Menu 12-9
Figure 12-6. ReturnNumArray.vi Diagram 12-11
Figure 12-7. Numeric Array Report with Graph 12-12
Figure 12-8. Numeric Array Report with Graph 12-16
Figure 12-9. Numeric Array Report with Graph 12-20
Figure 12-10. Callbacks Dialog Box 12-21
Figure 12-11. ModifyReportHeader in Sequence View 12-21
Figure 12-12. Insert String Local in ModifyReportHeader 12-22
Figure 12-13. Completed ModifyReportHeader Sequence 12-23
Figure 12-14. New HTML Header 12-24

Tables

Table 1-1. TestStand Subdirectories 1-2

Table 5-1. First-Level Properties of the Sequence Context 5-10

Table 6-1. Parameter Control Table of Values 6-39

1 Introduction to TestStand

This chapter contains instructions for installing TestStand, and provides an overview of the TestStand product.

TestStand is a flexible, powerful test executive framework for building, customizing and deploying a full-featured test executive system.

Installing TestStand

Before starting on your test applications, you must install TestStand on your computer. The TestStand setup program installs the software in approximately five minutes.

Minimum System Requirements

To run TestStand for Windows, you must have the following:

- Windows NT 4.0 or later, or Windows 98/95
- Personal computer using at least a 266 MHz Pentium class or higher microprocessor
- SVGA resolution (or higher) video adapter, minimally 800 × 600 video resolution
- Minimum of 64 MB of memory
- 100 MB of free hard disk space (250 MB recommended)
- Microsoft-compatible mouse

Installing TestStand

Follow these instructions to install TestStand:

1. Verify that your computer and monitor are powered on and that you have installed Windows NT 4.0 or later, or Windows 98/95.
2. Close all open Windows applications, and leave the operating system in Windows.
3. Insert the installation CD into the CD-ROM Drive.
4. Choose the **Run** option from the Desktop Taskbar.

5. Type `x:\tssetup.exe` (where *x* is the drive you are using) in the command line box and click on **OK**.
6. Follow the instructions that appear in the dialog boxes.

National Instruments recommends you install the complete TestStand program to take full advantage of all the TestStand capabilities. If you choose to install with options, select the options you want and follow the directions on the screen. If necessary, you can run the setup program again later and install additional files.

Caution If you have LabVIEW VIs that you saved with a version of LabVIEW older than 5.1 that call the TestStand API, you must mass compile them in LabVIEW 5.1 or later before installing TestStand 2.0. If you do not do mass compile your VIs, you will have to manually replace every ActiveX diagram node that uses the TestStand API. The TestStand 2.0 installer displays a message box if it detects an existing LabVIEW installation.

What the Setup Programs Install

The setup program installs the TestStand development environment and a number of additional files on your system. The full installation includes example files that illustrate many of the features in TestStand and tutorial programs that you use throughout this manual. The installer installs TestStand and the associated files in subdirectories on your hard disk as shown in Table 1-1.

Table 1-1. TestStand Subdirectories

Directory Name	Contents
`AdapterSupport`	Support files for the LabVIEW and C/CVI Standard Prototype Adapters
`Api`	TestStand ActiveX Automation Server libraries for LabWindows/CVI and MFC
`Bin`	TestStand sequence editor executable, engine DLLs, and support files
`Cfg`	Configuration files for TestStand engine and sequence editor options
`CodeTemplates`	Source code templates for step types—This directory contains an `NI` and a `User` subdirectory
`Components`	Components that come with TestStand and components that you develop—This directory includes callback files, converters, icons, language files, process model files, step types, source files, and utility files. It also contains an `NI` and a `User` subdirectory.

Table 1-1. TestStand Subdirectories (Continued)

Directory Name	Contents
`Doc`	Documentation files
`Examples`	Example sequences and tests
`OperatorInterfaces`	LabVIEW, LabWindows/CVI, Microsoft Visual Basic, and Delphi operator interfaces with source code. This directory contains an `NI` and a `User` subdirectory.
`Setup`	TestStand Installer/Uninstaller
`Tutorial`	Sequences and code modules that you use in the tutorial sessions in this manual

Learning TestStand

The best way to familiarize yourself with TestStand is to do the following:

1. Thoroughly read the `doc\readme.txt` file distributed with TestStand.
2. Read the remainder of this chapter for an overall idea of the concepts and capabilities of TestStand.
3. Complete the tutorial sessions in Chapters 2–12 as outlined in this manual.
4. Read Chapter 1, *TestStand Architecture Overview*, of the *TestStand User Manual*, and familiarize yourself with the other chapters in that manual.

Beginners should complete the tutorials in this manual. The *TestStand User Manual* generally assumes familiarity with *Getting Started with TestStand*. However, it is still useful to make quick references to other manuals and the online help as questions arise while you learn and use TestStand.

In Chapter 2, *Loading and Running Sequences*, you learn about the windows, menus, commands, and dialog boxes in TestStand. The *TestStand User Manual* contains a chapter devoted to each of the windows and components in TestStand. Scan these chapters for answers to any questions you might have as you use *Getting Started with TestStand*. The table of contents and index of each manual lists the location of helpful information in that manual.

The tutorial begins with a general introduction to the TestStand sequence editor and continues with sections devoted to building sequences in TestStand. Because each step of the tutorial builds on previous elements, you should follow the outline as given and not skip ahead.

TestStand System Overview

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

- Out-of-the-box configuration and components that give you a full-featured test executive that is ready to run.
- Numerous ways for you to modify the out-of-the-box configuration and components or to add new components. These extensibility mechanisms enable you to create the test executive that meets your particular requirements without modifying the TestStand test execution engine. You can upgrade to newer versions of TestStand without losing your customizations.
- Sophisticated sequencing, execution, and debugging capabilities and a powerful sequence editor that is separate from the run-time execution operator interfaces.
- Four separate run-time execution operator interfaces with source code for LabVIEW, LabWindows/CVI, Microsoft Visual Basic, and Delphi.
- Independence from particular Application Development Environments (ADEs). You can create test modules in a wide variety of ADEs and call preexisting modules or executables. You can create your own run-time execution operator interface in any language that can control ActiveX Automation Servers.
- Conversion of sequence files from the LabVIEW Test Executive Toolkit Version 5.0 or the LabWindows/CVI Test Executive Toolkit Version 2.0 to TestStand.
- Comprehensive ActiveX Application Programming Interface (API) for building multithreaded test executives and other sequencing applications.

The remainder of this chapter discusses the major software components of TestStand as a product.

Major Software Components of TestStand

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

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

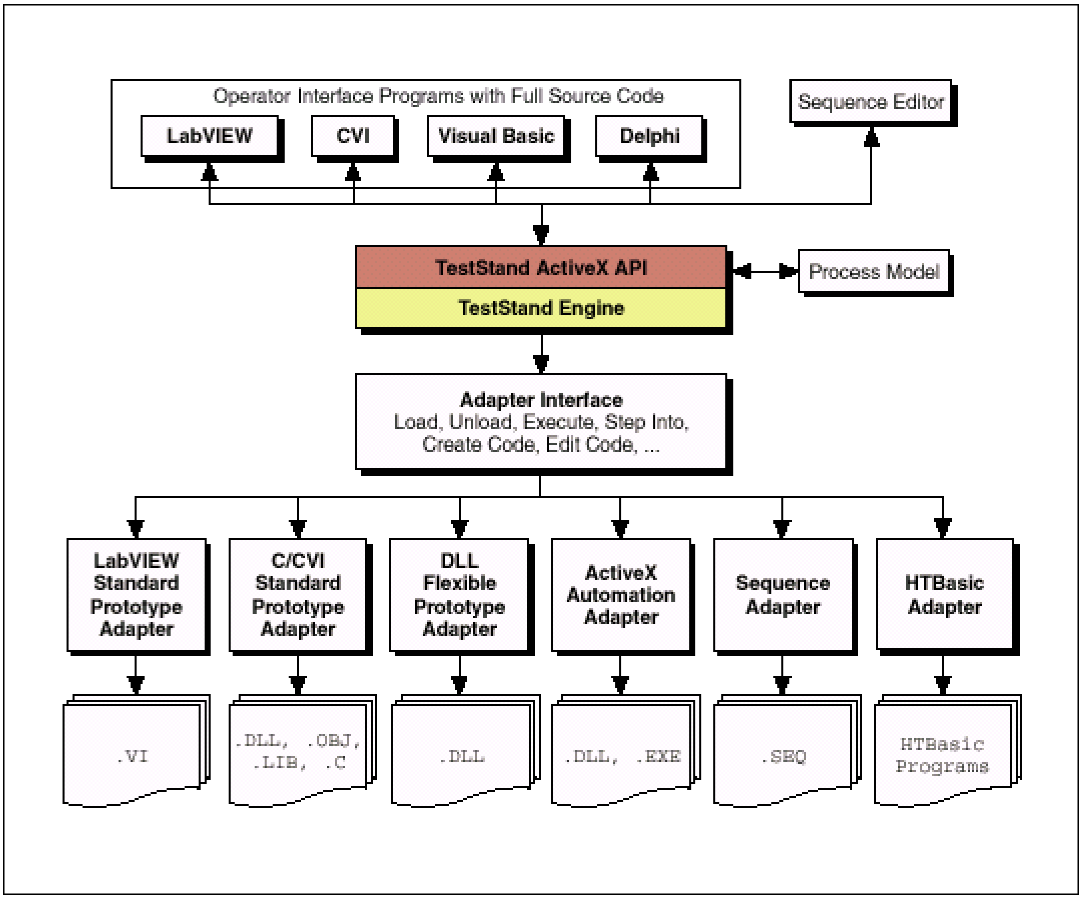

Operator Interface Programs with Full Source Code
LabVIEW
CVI
Visual Basic
Delphi
Sequence Editor
TestStand ActiveX API
TestStand Engine
Process Model
Adapter Interface
Load, Unload, Execute, Step Into,
Create Code, Edit Code, ...
LabVIEW Standard Prototype Adapter
C/CVI Standard Prototype Adapter
DLL Flexible Prototype Adapter
ActiveX Automation Adapter
Sequence Adapter
HTBasic Adapter
.VI
.DLL, .OBJ, .LIB, .C
.DLL
.DLL, .EXE
.SEQ
HTBasic Programs

Figure 1-1. TestStand System Architecture

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

TestStand Sequence Editor

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

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

TestStand Run-Time Operator Interfaces

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

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

Refer to Chapter 16, *Run-Time Operator Interfaces,* in the *TestStand User Manual* for more information on how to customize a run-time operator interface.

TestStand Test Executive Engine

The TestStand Test Executive Engine is a set of DLLs that export an ActiveX Automation API for creating, editing, executing, and debugging sequences. The TestStand sequence editor and run-time operator interfaces use the engine API. You can call the engine API from any programming environment that supports access to ActiveX Automation servers. Thus, you can call the engine API from test modules, including test modules you write in LabVIEW and LabWindows/CVI.

The documentation for the engine API is available as online help. You can access the online help through the **Help** menu of the sequence editor or from the TestStand program group.

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 the code module, how to call the code module, and how to pass parameters to the code module. The different types of code modules include LabVIEW VIs, C functions in DLLs, HTBasic files, HTBasic subroutines, and C functions in source, object, or library modules that you create in LabWindows/CVI or other compilers. Also, TestStand must know the list of parameters that the code module requires.

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

- **DLL Flexible Prototype Adapter**—Call C functions in a DLL with a variety of parameter types.
- **LabVIEW Standard Prototype Adapter**—Call any LabVIEW VI that has the TestStand standard G parameter list.
- **C/CVI Standard Prototype Adapter**—Call any C function that has the TestStand standard C parameter list. The function can be in an object file, library file, or DLL. The function can also be in a source file that is in the project you are currently using in the LabWindows/CVI development environment.
- **Sequence Adapter**—Call subsequences with parameters.
- **ActiveX Automation Adapter**—Call methods and access the properties of a ActiveX object.
- **HTBasic Adapter**—Calls HTBasic subroutines with no parameters. TestStand and HTBasic exchange data using the TestStand API. TestStand supports HTBasic version 7.2 or later.

The module adapters contain other important information besides the calling convention and parameter lists. If the module adapter is specific to an Application Development Environment (ADE), the adapter knows how to bring up the ADE, how to create source code for a new code module in the ADE, and how to display the source for an existing code module in the ADE. The DLL Flexible Prototype Adapter can query a DLL type library for the parameter list information and display the information to the sequence developer.

Process Models

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

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

TestStand provides a mechanism for defining a process model. A process model is in the form of a sequence file. You can edit a process model just as you edit your other sequences. TestStand ships with three fully functional process models: 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.

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.

For example, the default TestStand process model, `SequentialModel.seq`, provides two entry points: `Test UUTs` and `Single Pass`. The `Test UUTs` entry point initiates a loop that repeatedly identifies and tests UUTs. The `Single Pass` entry point tests a single UUT without identifying it. Both entry points provide the option to log results and produce reports. The entry points are called *execution entry points*. Execution entry points appear in the **Execute** menu of the sequence editor or operator interface when the active window is a sequence file window and the sequence file contains a sequence called `MainSequence`.

Use the Batch process model to control a set of test sockets that test multiple UUTs as a group. For example, you might 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. For example, you can specify that, because a step applies to the batch as a whole, the step runs only once per batch instead of once for each UUT. The Batch model also makes it easy 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. Finally, the Batch model can generate batch reports that summarize the test results for the UUTs in the batch.

Use the Parallel model to control multiple independent test sockets. The Parallel model allows you to start and stop testing on any socket at any time. For example, you might have five test sockets for testing radios. The Parallel model allows you to load a new radio into an open socket while the other sockets are busy testing other radios.

The default process model is `SequentialModel.seq`. To change the process model, select **Configure»Station Options** and click on the Model tab. You can select a different process model from the Station Model ring or you can specify the model by clicking on the **Browse** button. You also can use the Sequence File Properties dialog box to specify that a sequence file always uses a particular process model. For more information about the process models, refer to Chapter 1, *TestStand Architecture Overview*, Chapter 3, *Configuring and Customizing TestStand*, and Chapter 14, *Process Models,* in the *TestStand User Manual*.

Proceed to the next chapter to begin the TestStand tutorials.

2

Loading and Running Sequences

In this chapter, you learn to load and run sequences in the TestStand sequence editor, and you learn about the windows in the sequence editor.

Starting TestStand

To start TestStand, complete the following steps:

1. Launch TestStand by going to the **Start** system menu, and selecting **Programs»National Instruments»TestStand»Sequence Editor**. After the sequence editor displays a main window, the Login dialog box appears, as shown in Figure 2-1.

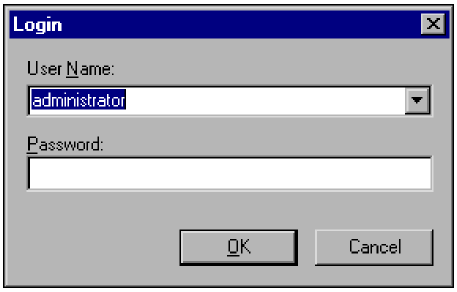


Figure 2-1. Login Dialog Box

2. If the default User Name is not `administrator` as shown in Figure 2-1, click the User Name control and select `administrator` from the popup list.

3. The default password for the `administrator` user login is empty, so click on the **OK** button without entering a password. Figure 2-2 shows the main window for the sequence editor.

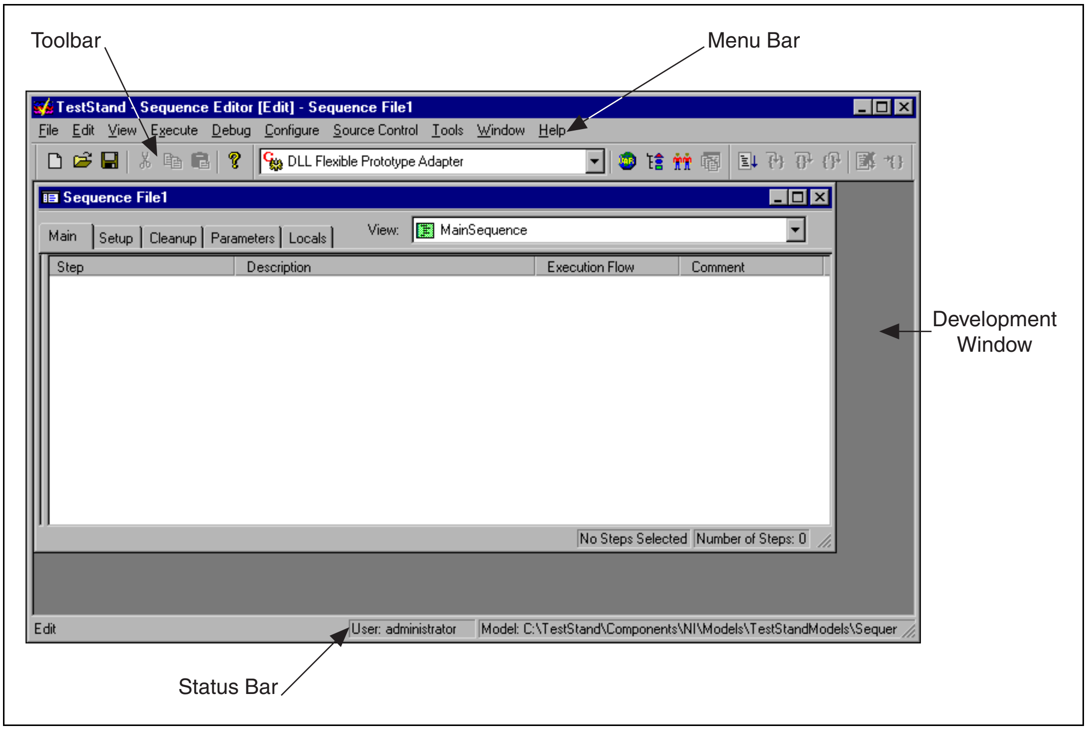

Toolbar
Menu Bar
TestStand - Sequence Editor [Edit] - Sequence File1
File Edit View Execute Debug Configure Source Control Tools Window Help
DLL Flexible Prototype Adapter
Sequence File1
Main Setup Cleanup Parameters Locals
View: MainSequence
Step Description Execution Flow Comment
No Steps Selected Number of Steps: 0
Development Window
Edit
User: administrator
Model: C:\TestStand\Components\NI\Models\TestStandModels\Sequer
Status Bar

Figure 2-2. Sequence Editor Main Window

Introduction to the Sequence Editor

The sequence editor window has four main parts: the menu bar, the toolbar, the development workspace, and the status bar. A detailed discussion of each of these parts is presented in Chapter 2, *Sequence Editor Concepts*, in the *TestStand User Manual*.

Menu Bar

The menu bar contains the following menus: **File**, **Edit**, **View**, **Execute**, **Debug**, **Configure**, **Source Control**, **Tools**, **Windows**, and **Help**. Browse the menus in the sequence editor to familiarize yourself with their contents. Chapter 4, *Sequence Editor Menu Bar,* in the *TestStand User Manual* contains a detailed explanation of each menu item.

Toolbar

The toolbar contains shortcuts to commonly used selections of the menu bar. As shown in Figure 2-3, the toolbar contains three sections: standard, debug, and environment.

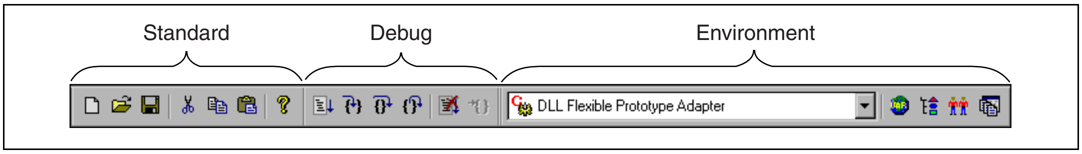


Figure 2-3. Sequence Editor Toolbar

- **Standard Section**—Contains buttons for creating, loading, saving, cutting, and pasting sequence files.
- **Debug Section**—Contains buttons for executing a sequence, stepping into, stepping over, stepping out, pausing, and terminating execution.
- **Environment Section**—Contains the adapter selection ring, buttons for opening other TestStand station windows, and a button to bring an open workspace to the front.

Development Workspace

The development workspace is the main area of the sequence editor. In this area the sequence editor displays its windows.

Status Bar

The status bar displays common information in the sequence editor. As shown in Figure 2-4, the status bar contains three sections: selection help, login display, and model display.

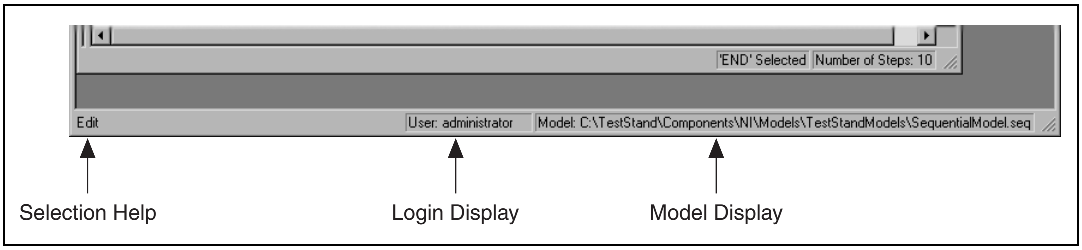


Figure 2-4. Sequence Editor Status Bar

- **Selection Help**—Displays information on the currently selected menu item.
- **Login Display**—Displays the user name of the current user.
- **Model Display**—Shows the pathname of the process model file.

You can manipulate all windows in TestStand through TestStand menu selections or through the standard means for manipulating windows on the operating system. For example, you can close, maximize, minimize, and position TestStand windows on the screen through any of the Windows standard windowing methods.

Loading a Sequence File

To view the features of the TestStand sequence editor, you must first load a sequence file into the TestStand sequence editor. To do this, follow these instructions.

1. Select **File»Open**. When you make this selection, an Open dialog box appears as shown in Figure 2-5.

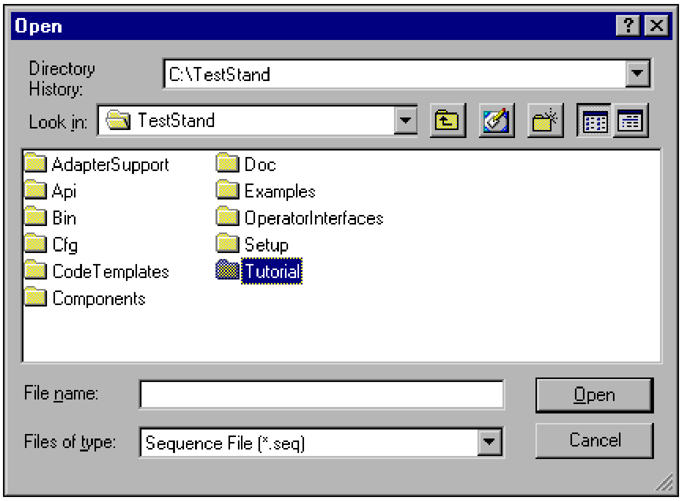
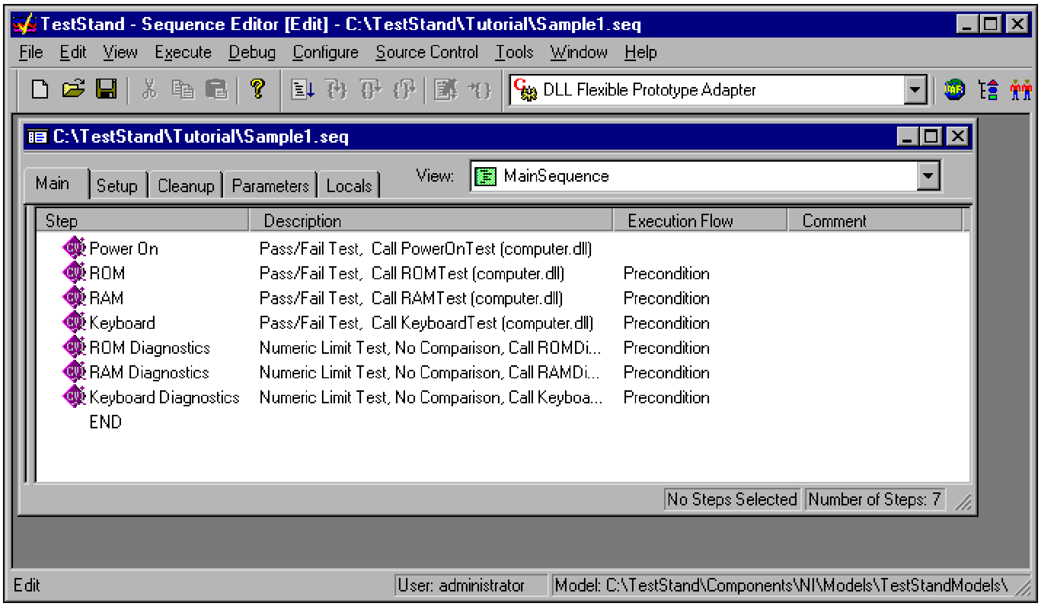


Figure 2-5. Open Dialog Box

2. Navigate to the `TestStand\Tutorial` subdirectory.
3. Select the `Sample1.seq` sequence file from the `Tutorial` subdirectory, and click on the **Open** button.

After you open the sequence file, a new sequence file window appears in the sequence editor, as shown in Figure 2-6.


TestStand - Sequence Editor [Edit] - C:\TestStand\Tutorial\Sample1.seq
File Edit View Execute Debug Configure Source Control Tools Window Help
DLL Flexible Prototype Adapter
C:\TestStand\Tutorial\Sample1.seq
Main Setup Cleanup Parameters Locals
View: MainSequence

Step	Description	Execution Flow	Comment
Power On	Pass/Fail Test, Call PowerOnTest (computer.dll)		
ROM	Pass/Fail Test, Call ROMTest (computer.dll)	Precondition	
RAM	Pass/Fail Test, Call RAMTest (computer.dll)	Precondition	
Keyboard	Pass/Fail Test, Call KeyboardTest (computer.dll)	Precondition	
ROM Diagnostics	Numeric Limit Test, No Comparison, Call ROMDi...	Precondition	
RAM Diagnostics	Numeric Limit Test, No Comparison, Call RAMDi...	Precondition	
Keyboard Diagnostics	Numeric Limit Test, No Comparison, Call Keyboa...	Precondition	
END			

No Steps Selected Number of Steps: 7

Edit User: administrator Model: C:\TestStand\Components\NI\Models\TestStandModels\

Figure 2-6. Sample1.seq Sequence File Window

Note If you are not the first person to use this tutorial on your computer, it might be necessary to reinstall TestStand to get the unmodified versions of the tutorial files.

If other people will be using the Getting Started example files on your computer, be sure to use the **Save As** option to save your files under different file names. When you must save a file, this manual specifies the suggested name.

The `Sample1.seq` sequence file is a simulated test of your computer in which you can choose various hardware components to "fail" the test. The sequence runs tests that are functions in a dynamic link library (`.dll`) written with LabWindows/CVI.

The sequence file appears as a separate window within the sequence editor. This window is called a *Sequence File window*. You can load multiple sequence files into the sequence editor, and the sequence editor displays each in its own Sequence File window.

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

Figure 2-7 shows the contents of the View ring for the `Sample1.seq` sequence file.

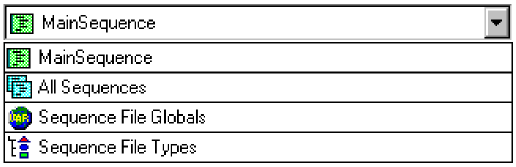


Figure 2-7. Sequence File View Ring

4. Select `MainSequence` in the View ring if it is not already selected.

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

The Main, Setup, and Cleanup tabs each show one of the step groups in the sequence. You can view the contents of each tab by clicking on the tab. Following are the purposes of the steps you insert in each step group:

- **Main**—Test your UUT.
- **Setup**—Initialize or configure your instruments, fixtures, and UUT.
- **Cleanup**—Power down or uninitialize your instruments, fixtures, and UUT.

View the contents of each tab and return to the Main step group when you finish.

About Sequences

A sequence consists of a series of steps. In TestStand, a step can do many things, such as initializing an instrument, performing a complex test, or making a decision that affects the flow of execution in a sequence. A step can jump to another step, call a subsequence, call an external code module, or change the value of a variable or property.

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

Sequences can have any number of local variables. You can use local variables to hold values that steps use. You can also use local variables for maintaining counts, for holding intermediate values, or for any other value storage. You define the local variables for a sequence in the Locals tab.

Running a Sequence

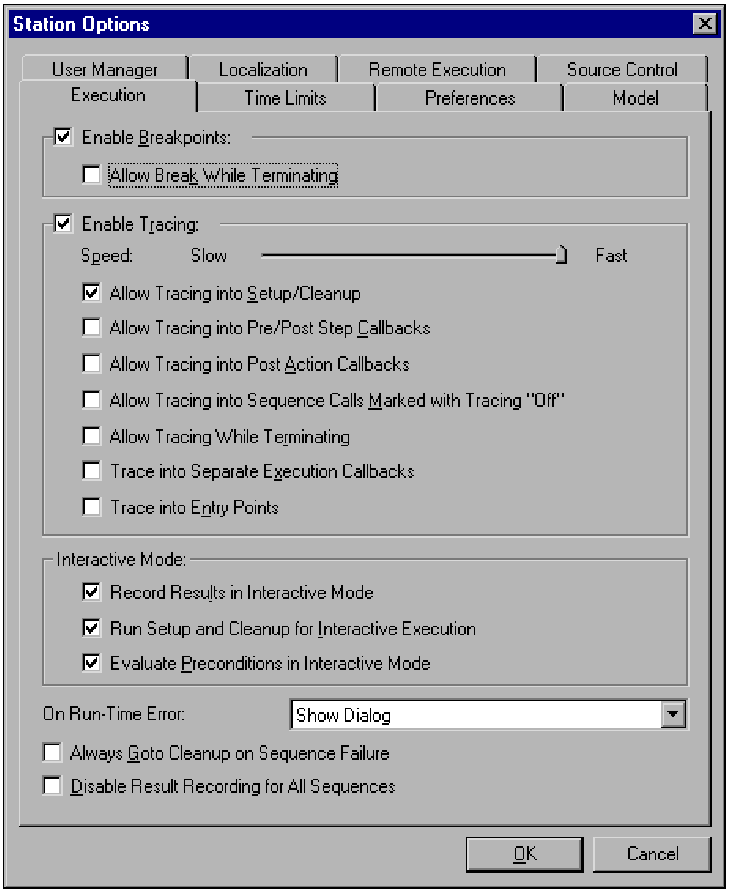
When you run a sequence in the sequence editor, you are initiating an *execution*. You will examine two methods of running a sequence: running a sequence directly and running a sequence using a *process model*, which is a special type of sequence for directing the high-level sequence flow.

Setting Up Tracing Options

Before you run a sequence, confirm that the sequence editor tracing options are configured properly by completing the following steps.

1. Select **Configure»Station Options**, which displays the Station Options dialog box.

2. Confirm that the options on the Execution Tab are set, as shown in Figure 2-8. Update any settings that are different.



Station Options

- [✓] Enable Breakpoints:
  - [ ] Allow Break While Terminating
- [✓] Enable Tracing:
  - Speed: Slow — Fast
  - [✓] Allow Tracing into Setup/Cleanup
  - [ ] Allow Tracing into Pre/Post Step Callbacks
  - [ ] Allow Tracing into Post Action Callbacks
  - [ ] Allow Tracing into Sequence Calls Marked with Tracing "Off"
  - [ ] Allow Tracing While Terminating
  - [ ] Trace into Separate Execution Callbacks
  - [ ] Trace into Entry Points
- Interactive Mode:
  - [✓] Record Results in Interactive Mode
  - [✓] Run Setup and Cleanup for Interactive Execution
  - [✓] Evaluate Preconditions in Interactive Mode
- On Run-Time Error: Show Dialog
- [ ] Always Goto Cleanup on Sequence Failure
- [ ] Disable Result Recording for All Sequences

Figure 2-8. Execution Tab on the Station Options Dialog Box

3. Click on **OK** to close the dialog box.

Running a Sequence Directly

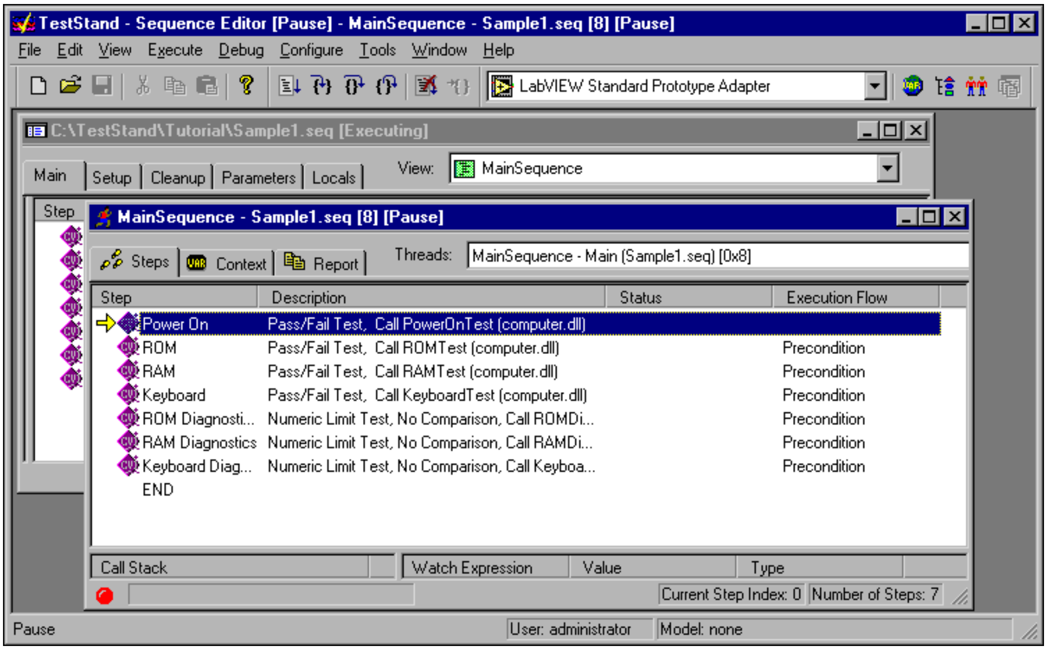
The easiest way to start a sequence execution is to run a sequence directly.

Follow these steps to run `MainSequence` in the `Sample1.seq` sequence file window:

1. Select `MainSequence` in the View ring of the sequence file window if it is not already selected.
2. Select **Execute»Run MainSequence**.

   When you make this selection, the sequence editor opens a new window. This window is called an *Execution window*. In an Execution window, you can view steps as they execute, the values of variables and properties, and the test report when the execution completes.

   Figure 2-9 shows an example Execution window.


TestStand - Sequence Editor [Pause] - MainSequence - Sample1.seq [8] [Pause]
File Edit View Execute Debug Configure Tools Window Help
C:\TestStand\Tutorial\Sample1.seq [Executing]
Main Setup Cleanup Parameters Locals
MainSequence - Sample1.seq [8] [Pause]
Steps Context Report
Call Stack
Watch Expression Value Type

LabVIEW Standard Prototype Adapter

View: MainSequence

Threads: MainSequence - Main (Sample1.seq) [0x8]

Step	Description	Status	Execution Flow
Power On	Pass/Fail Test, Call PowerOnTest (computer.dll)		
ROM	Pass/Fail Test, Call ROMTest (computer.dll)		Precondition
RAM	Pass/Fail Test, Call RAMTest (computer.dll)		Precondition
Keyboard	Pass/Fail Test, Call KeyboardTest (computer.dll)		Precondition
ROM Diagnosti...	Numeric Limit Test, No Comparison, Call ROMDi...		Precondition
RAM Diagnostics	Numeric Limit Test, No Comparison, Call RAMDi...		Precondition
Keyboard Diag...	Numeric Limit Test, No Comparison, Call Keyboa...		Precondition
END			

Current Step Index: 0 Number of Steps: 7

Pause User: administrator Model: none

Figure 2-9. Sample1.seq Execution Window

After the execution starts, a Test Simulator dialog box appears in front of the Execution window, as shown in Figure 2-10.

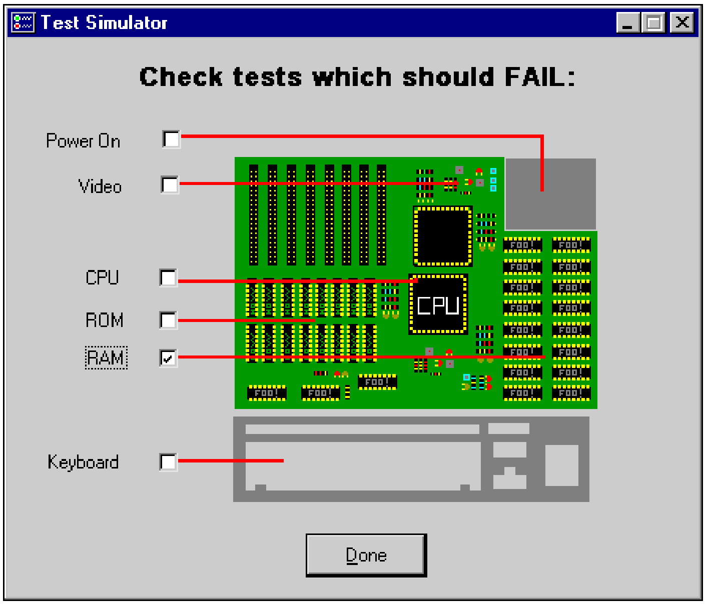


Figure 2-10. Test Simulator Dialog Box

One of the steps in the execution displays this dialog box. The dialog box prompts you to enter which computer component, if any, you want to “fail” during the execution.

3. Select the RAM test by clicking on its checkbox.
4. Click on the **Done** button. Observe the Execution window as it traces through the steps that TestStand runs.

   The sequence editor displays the progress of an execution by placing a yellow pointer icon to the left of the currently executing step in the Steps tab. The pointer icon is called the *execution pointer*. Notice that the status column for the RAM test contains the value `Failed`. When the execution completes, the status section of the window title changes from [Running] to [Completed], and the Execution window dims.

5. After the execution completes, close the Execution window by selecting **File»Close**, by selecting **Windows»Close Completed Execution Displays**, or by clicking on the X icon on the window title bar.

If you want to rerun the sequence, repeat steps 1 through 5 above. For step 3, select any test other than the Video or CPU test to fail in the Test Simulator dialog box. You will complete the Video and CPU tests in a subsequent exercise.

Note If you are using a fast computer and you want to slow down the tracing feature of TestStand, you can change the Speed slider control value on the Execution tab of the Station Options dialog box.

Running a Sequence Using the Sequential Process Model

In addition to executing a sequence directly, you can execute a sequence using a process model entry point. Chapter 1, *Introduction to TestStand*, explains that an entry point is simply a sequence in a process model sequence file. When you execute an entry point, it performs a series of operations before and after calling the `MainSequence` of your sequence file. Common operations of the process model are identifying the UUT, notifying the operator of pass/fail status, generating a test report, and logging results.

Follow these steps to run `MainSequence` in the `Sample1.seq` sequence file using the `Single Pass` execution entry point of `SequentialModel.seq`.

1. Set `SequentialModel.seq` as your default process model by selecting **Configure»Station Options**. Click on the Model tab and select the sequential model from the Station Model ring control.
2. Verify that `Sample1.seq` sequence file window is the active window.
3. Select **Execute»Single Pass**.
4. Once again, select any test other than the Video or CPU test to fail in the Test Simulator dialog box. You will complete the Video and CPU tests in a subsequent exercise.
5. Click on **Done** to close the prompt.

   Notice that after TestStand executes the steps in the main sequence, the `Single Pass` entry point generates a test report. While TestStand generates the report, a status indicator bar appears at the bottom of the Execution window, as shown in Figure 2-11.

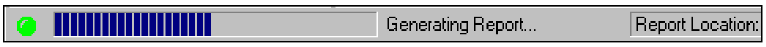


Figure 2-11. Report Generating Status Bar

6. After TestStand generates the report, the Execution window displays the report in the report tab. Examine the test report and notice that it contains information on the results of each step TestStand executes.

   You learn more about the test report feature in Chapter 12, *Customizing the Report*, in this manual.
7. After the execution completes, close the Execution window by selecting **File»Close**, by selecting **Windows»Close Completed Execution Displays**, or by clicking on the X icon on the window title bar.
8. Select **Execute»Test UUTs**.

   Before executing the steps in the main sequence, the process model sequence displays a UUT Information dialog box requesting a serial number.
9. Enter any number and click on the **OK** button.
10. Select a test to fail in the Test Simulator dialog box.
11. Click on **Done**. Observe the Execution window as the sequence is executing.

    After completing the steps in the main sequence, the process model displays a banner that indicates the result of the UUT.
12. Click on the **OK** button to close the UUT Result banner. The process model now generates a report but instead of completing the execution and displaying the report, the process model displays the UUT Information dialog box again.
13. Repeat steps 9 through 12 for several different serial numbers.
14. Click on the **Stop** button to stop the loop and complete the execution.

    After the execution completes, TestStand displays a test report for all of the UUTs.
15. Examine the test report and verify that it has indeed recorded the results for each UUT.
16. After the execution completes, close the Execution window by selecting **File»Close**, by selecting **Windows»Close Completed Execution Displays**, or by clicking on the X icon on the window title bar.

Running a Sequence Using the Batch Process Model

The batch process model executes the same test sequence on multiple UUTs at the same time. With the batch process model, you start and finish testing all UUTs in a set at the same time. The batch model also provides batch synchronization features. For example, you can specify that, because a step applies to the batch as a while, the step runs only once per batch instead of once for each UUT. The batch model 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 also can generate batch reports that summarize the test results for the UUTs in the batch.

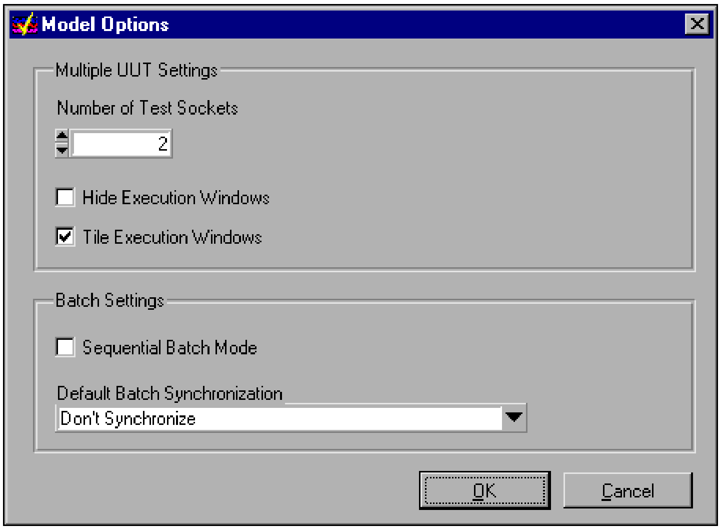
Follow these steps to run the `BatchUUT.seq` sequence file using the `Single Pass` execution entry point of `BatchModel.seq`.

1. Open the sequence file `<TestStand>\Tutorial\BatchUUT.seq`.

Note You do not need to change your default process model for this exercise. The sequence file is configured to always use the batch process model, regardless of the default process model of the sequence editor. You use the Sequence File Properties dialog box to specify that a sequence file always uses a particular process model.

2. Select **Configure»Model Options**.
3. Change the settings to match those shown in Figure 2-12 and then click on the **OK** button.

   The number of Test Sockets is the number of UUTs to test in the batch. It is easier to interpret the executions if you start with a low number of Test Sockets.


Model Options

Multiple UUT Settings

Number of Test Sockets: 2

☐ Hide Execution Windows

☑ Tile Execution Windows

Batch Settings

☐ Sequential Batch Mode

Default Batch Synchronization: Don't Synchronize

Figure 2-12. Model Options Dialog Box

4. Select **Execute»Single Pass**.

   During the execution, TestStand displays execution windows for each Test Socket that show the tracing for each execution. There are several synchronization sections within this example. The first section simulates temperature change in a chamber. The step that performs the test is only executed once per batch instead of once per UUT. The second section simulates sharing a common resource, a pulse generator. Only one UUT at a time executes steps in this section. The third section simulates a frequency sweep, which all UUTs execute in parallel. Finally a message popup displays the time it takes to execute the frequency sweeps. Only one UUT reports the time.

   TestStand generates a report for the batch and for each UUT. The batch report summarizes the results for all the UUTs in the batch. When the report format is HTML, the batch report provides hyperlinks to each UUT report. Click on the hyperlinks within the batch report to view specific UUT results.

5. After the execution completes, close the Execution window by selecting **File»Close**, by selecting **Windows»Close Completed Execution Displays**, or by clicking on the X icon on the window title bar.

6. Select **Execute»Test UUTs**.

   Before executing the steps in the main sequence, the process model sequence displays a UUT Information dialog box requesting a batch serial number and UUT serial numbers for each Test Socket. Notice that you can disable particular Test Sockets.

7. Enter any batch serial number and UUT serial numbers and then click on the **Go** button.

8. Click on the **View Batch Report** button.

   After completing the steps in the main sequence, the process model displays a banner that indicates the result of the UUTs. This banner allows you to view the batch report and the individual UUT reports.

   Notice that you have the option to view the Entire File or Current Only. The Entire File consists of all tested batches while the Current Only consists of the most recent batch that was tested.

9. Select **Current Only**.

   TestStand launches an external viewer to display the reports. By default, the HTML report displays in Microsoft Internet Explorer and the text reports display in Microsoft Notepad. You can configure your system to use other applications as external viewers.

10. Close the external viewer and return to the result banner.
11. Click the **Continue** button.
12. Repeat steps 7 through 11 for several different batches.
13. Click on the **Stop** button to complete the execution.

    After the execution completes, TestStand displays test reports for all batches and UUTs in an internal viewer.
14. Examine the reports and notice that TestStand records results for each batch and UUT.
15. After the execution completes, close the Execution window by selecting **File»Close**, by selecting **Windows»Close Completed Execution Displays**, or by clicking on the X icon on the window title bar.

This concludes this tutorial session. In the next session, you learn how to add steps to a sequence and edit step properties.

3 Editing Steps in a Sequence

In this chapter, you add a step to a sequence and then configure the properties of the step. Also, you add a subsequence call to another sequence.

Setting Up the Example

If you did not directly proceed from Chapter 2, *Loading and Running Sequences*, follow these steps to set up the TestStand sequence editor so you can complete this tutorial session.

1. Close all windows in the sequence editor.
2. Select **File»Open**, open the `Sample1.seq` file in the `Teststand\Tutorial` directory.
3. Display the `MainSequence` sequence in the Sequence File window by selecting `MainSequence` in the View ring.

Adding a New Step

TestStand contains a set of predefined types of steps. Step types define a list of standard properties and behaviors for each step of that type. Step types might call code modules using a module adapter or step types might not use module adapters at all.

The predefined step types available in TestStand include:

- Action
- Numeric Limit Test
- Multiple Numeric Limit Test
- String Value Test
- Pass/Fail Test
- Label
- Goto
- Statement
- Property Loader

- Message Popup
- Call Executable
- Call Sequence
- Synchronization (Lock, Semaphore, Rendezvous, Queue, Notification, Wait, Thread Priority, Batch Synchronization)
- Database (Open Database, Open SQL Statement, Close Database, Close SQL Statement, Data Operation)
- IVI (Power Supply, DMM, Scope, Fgen, Tools)

For a description of each of these step types, refer to Chapter 10, *Built-In Step Types*, in the *TestStand User Manual*.

In this exercise, you add a step to the sequence and configure that step to call a function in a DLL code module. Follow these steps to insert a Pass/Fail Test into the sequence:

1. Before you can insert a step that calls a code module, you must specify which module adapter the step uses. You can enter the selected module adapter by clicking on the Adapter Selection ring control on the toolbar as shown in Figure 3-1 or by using the **Configure»Adapters**. The selected adapter applies only to step types that can use any module adapter.

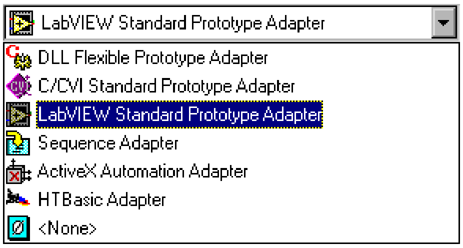
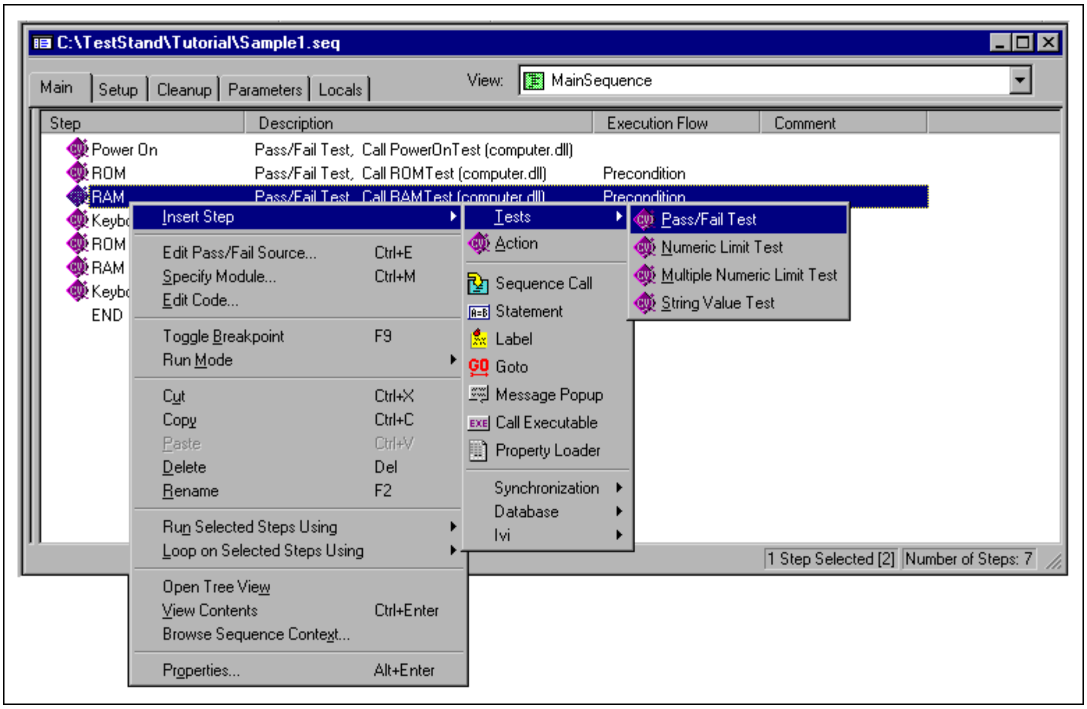

LabVIEW Standard Prototype Adapter
DLL Flexible Prototype Adapter
C/CVI Standard Prototype Adapter
LabVIEW Standard Prototype Adapter
Sequence Adapter
ActiveX Automation Adapter
HTBasic Adapter
<None>

Figure 3-1. Selecting the Module Adapter

   When you insert a step in a sequence, TestStand binds the step to the adapter that is currently selected in the ring on the sequence editor toolbar. If you choose <None> for the selected adapter and you insert a step, the step you insert does not call a code module. The icon for the adapter appears as the icon for the step.

   Select C/CVI Standard Prototype Adapter in the Adapter Selection Ring. The C/CVI Standard Prototype Adapter allows you to call any C function that has the TestStand standard C parameter list. The function can be in a dynamic linked library (`.dll`), source file (`.c`), object file (`.obj`), or static library file (`.lib`).

2. Right click on the RAM test in the Sequence File window and select **Insert Step»Tests»Pass/Fail Test** from the menu that appears, as shown in Figure 3-2. This menu is called a *context menu*.



View: MainSequence

Step	Description	Execution Flow	Comment
Power On	Pass/Fail Test, Call PowerOnTest (computer.dll)		
ROM	Pass/Fail Test, Call ROMTest (computer.dll)	Precondition	
RAM	Pass/Fail Test, Call RAMTest (computer.dll)	Precondition	
Keybo			
ROM			
RAM			
Keybo			
END			

1 Step Selected [2] Number of Steps: 7

Figure 3-2. Inserting a New Step

When you make this selection, the sequence editor inserts a new Pass/Fail Test step after the RAM step.

Normally, you 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 evaluates a Boolean expression to determine whether the step passes or fails.

3. By default, the new test is named Pass/Fail Test. After you insert the step, the name of the step is selected.

4. You can rename the new step by typing `Video Test` and pressing <Enter>. If you ever want to rename a step name later, you can right click on the step name and select the **Rename** command from the context menu.

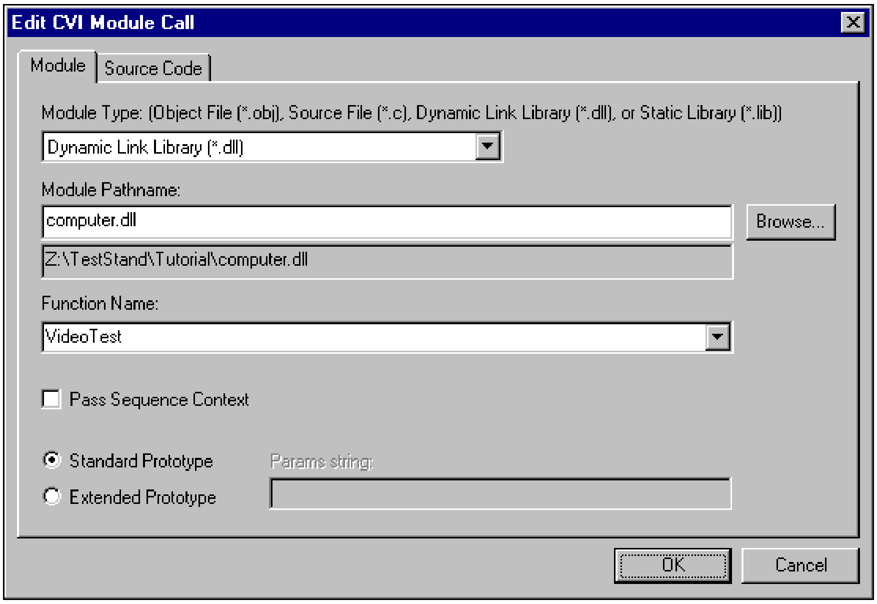
Specifying the Test Module

After you add a new step to the sequence, you must specify the test module that the step executes.

1. Right click on the new Video Test step and select the **Specify Module** command from the context menu.

   When you make this selection, the sequence editor displays a dialog box in which you can specify the code module for the step, along with any parameters to pass when invoking the code module. The actual title of the dialog box varies depending on the module adapter associated with the step.

   After you complete the required information in the dialog box, TestStand stores the information as properties of the step. For the C/CVI Standard Prototype Adapter, the sequence editor displays the Edit C/CVI Module Call dialog box, shown in Figure 3-3.
2. Select Dynamic Link Library (`*.dll`) in the Module Type ring control. This selection specifies that the code module for the test calls a function within a DLL.
3. Click on the **Browse** button to select the DLL the step calls.
4. Select the `computer.dll` file in the `TestStand\Tutorial` subdirectory. When you select a DLL, TestStand attempts to read the type library of the DLL and lists all the exported functions in the Function Name ring control.
5. Select the `VideoTest` function in the Function Name ring control by clicking on the arrow to the right of the control. You can click on the scroll bar arrows to scroll down to the `VideoTest` function. This function is a easy routine that returns a Boolean value to indicate whether the test passes or fails. Figure 3-3 shows the completed dialog box.



Module Type: (Object File (*.obj), Source File (*.c), Dynamic Link Library (*.dll), or Static Library (*.lib))
Dynamic Link Library (*.dll)
Module Pathname:
computer.dll
Z:\TestStand\Tutorial\computer.dll
Function Name:
VideoTest
☐ Pass Sequence Context
● Standard Prototype
○ Extended Prototype
Params string:

Figure 3-3. Specify Module Dialog Box

6. Click on the **OK** button to close the Edit C/CVI Module Call dialog box and return to the Sequence File window.

Changing Step Properties

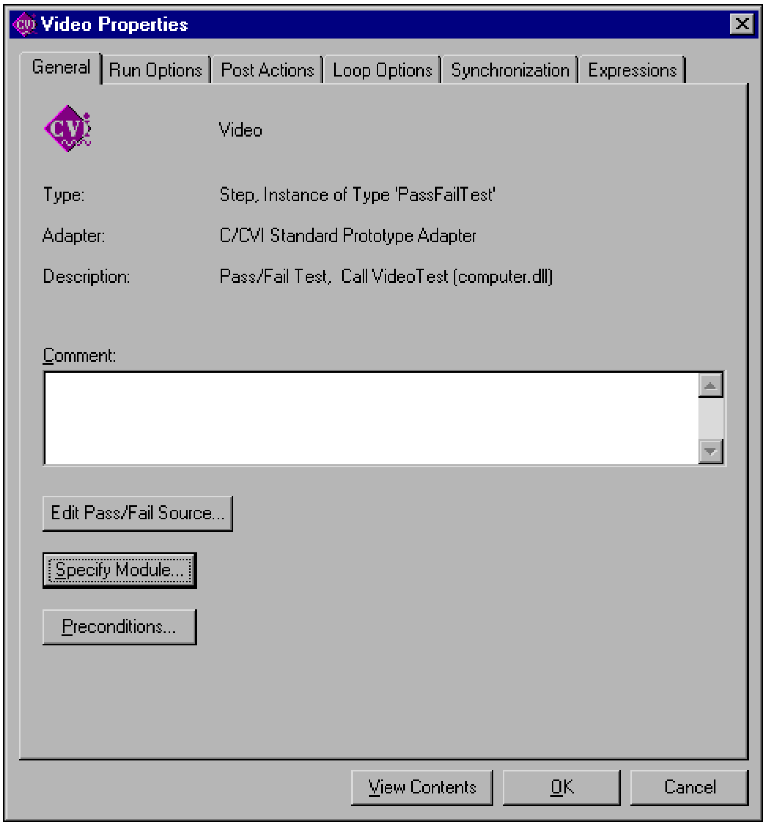
Each step in a sequence contains properties. The type of a step determines the set of properties that a step has. All steps have a common set of properties that determine the following:

- When to load the step
- When to execute the step
- What information TestStand examines to determine whether a test passes or fails
- Whether TestStand executes the step in a loop
- What conditional actions occur upon step completion

In this exercise, you examine these common properties and how you can set their values.

1. Right click on the Video Test and select the **Properties** command from the context menu.

   When you make this selection, TestStand displays the Step Properties dialog box for the step, shown in Figure 3-4.



Video

Type: Step, Instance of Type 'PassFailTest'

Adapter: C/CVI Standard Prototype Adapter

Description: Pass/Fail Test, Call VideoTest (computer.dll)

Figure 3-4. Step Properties Dialog Box

2. Click on the **Preconditions** button on the dialog box, which displays the Preconditions dialog box, shown in Figure 3-5.

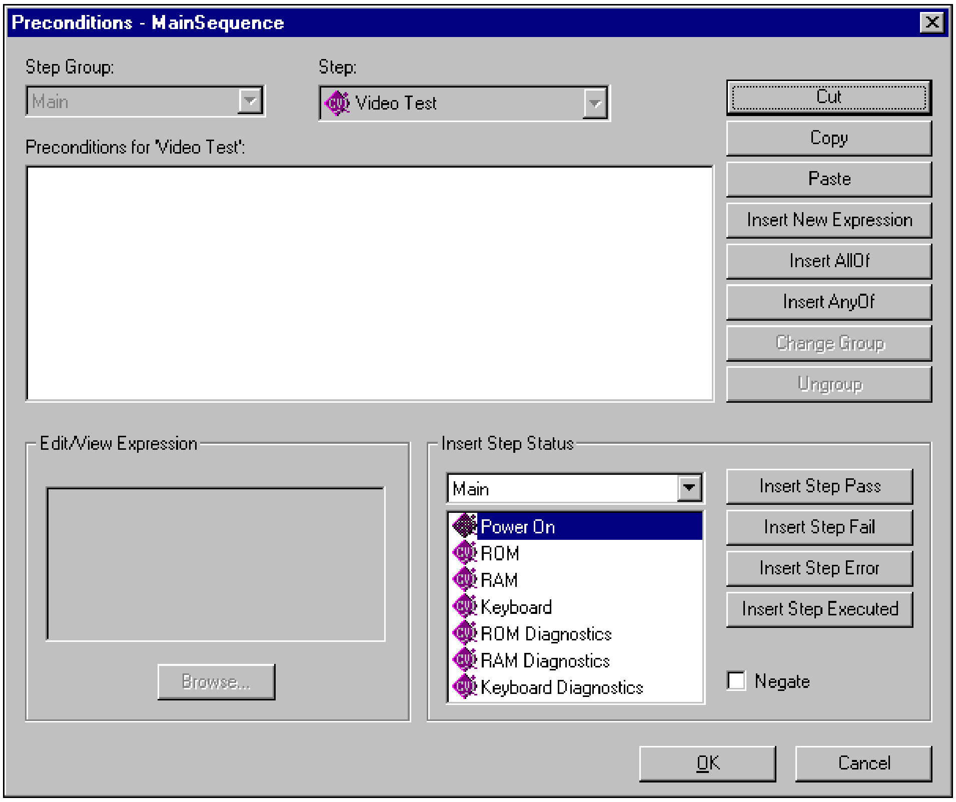


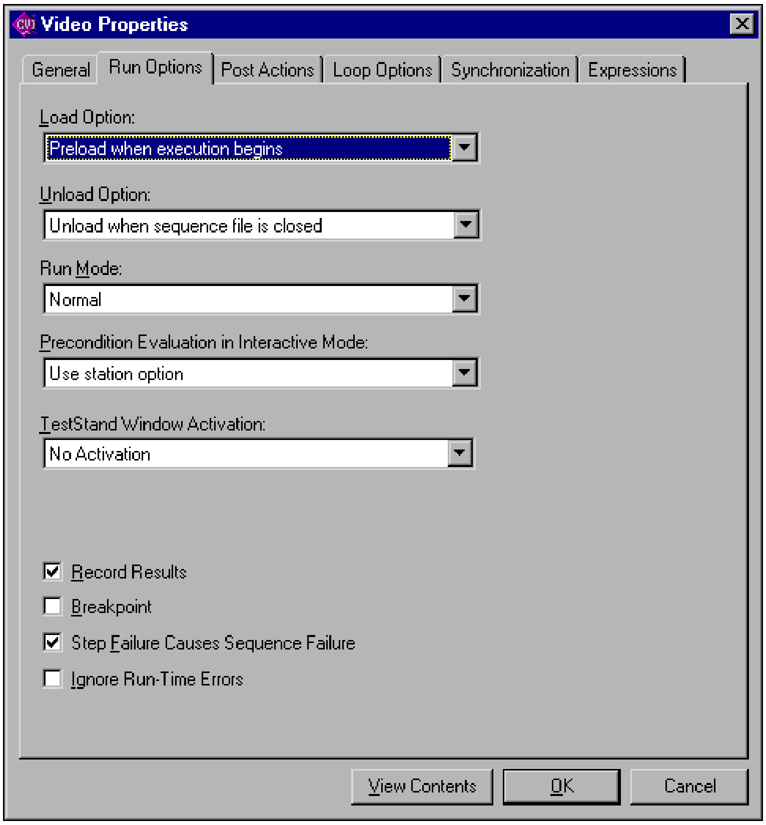
Figure 3-5. Preconditions Dialog Box

A precondition specifies the conditions that must evaluate to be true for TestStand to execute a step during the normal flow of execution in a sequence. For example, you might want to run a step only if a previous step passes.

3. For the Video Test, define a precondition so that the step executes only if the Power On step passes, as follows:
   a. Under the Insert Step Status section of the dialog box, click on the Power On step in the list of step names for the Main step group.
   b. Add a condition to the precondition list by clicking on the **Insert Step Pass** button. The Preconditions for 'Video Test' text box now contains the string `PASS Power On`, which indicates that the step executes only if the Power On step passes.

c. Click on the **OK** button to close the Preconditions dialog box and return to the Step Properties dialog box.

4. Click on the Run Options tab to display its tab, shown in Figure 3-6. This tab contains various settings that affect how TestStand runs this step.



Load Option: Preload when execution begins

Unload Option: Unload when sequence file is closed

Run Mode: Normal

Precondition Evaluation in Interactive Mode: Use station option

TestStand Window Activation: No Activation

[✓] Record Results

[ ] Breakpoint

[✓] Step Failure Causes Sequence Failure

[ ] Ignore Run-Time Errors

Figure 3-6. Run Options Tab

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

- **Load Option**—Specifies one of the following load option settings for the step.
  - **Preload when opening sequence file**—Loads the step module when TestStand loads into memory the sequence that contains the step.

- **Preload when execution begins**—Loads the step module when any sequence in the sequence file that contains the step begins executing. This value is the default setting.
- **Load dynamically**—Does not load the step module until the step is ready to call it.

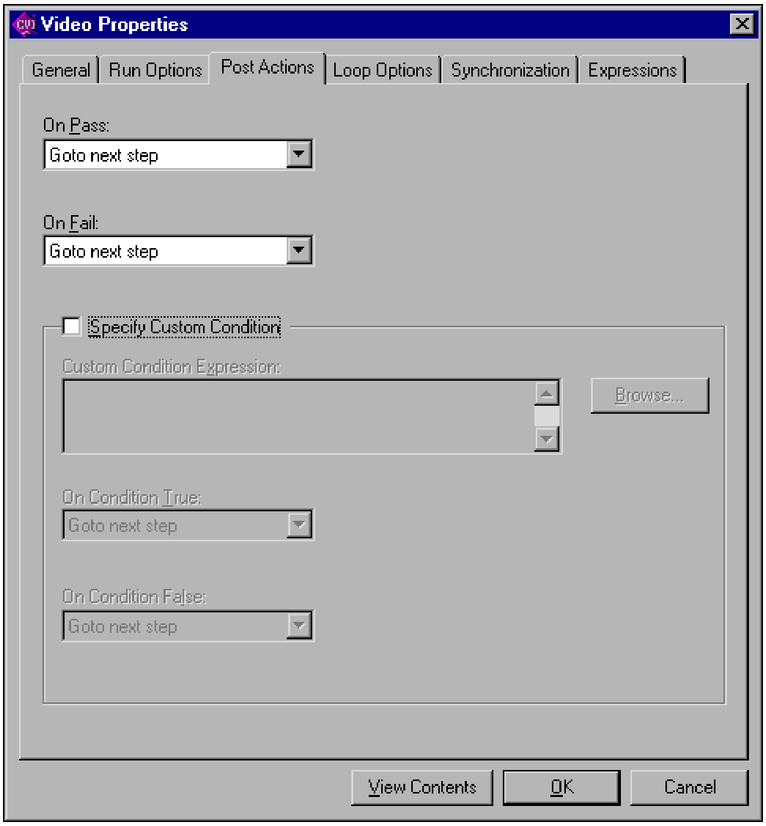
- **Unload Option**—Specifies one of the following Unload Option settings for the step.
  - **Unload when precondition fails**—Unloads the step module when the precondition for the step evaluates to `False`.
  - **Unload after step executes**—Unloads the step module after the step finishes executing.
  - **Unload after sequence executes**—Unloads the step module after the sequence that contains it finishes executing.
  - **Unload when sequence file is closed**—Unloads the step module when TestStand unloads the sequence file that contains the step from memory. This value is the default setting.

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

- **Run Mode**—Sets the following run-mode values for the step.
  - Force Pass
  - Force Fail
  - Skip
  - Normal
- **Precondition Evaluation in Interactive Mode**—Determines whether TestStand evaluates the step precondition when you run the step interactively. The ring control contains the following options:
  - Use Station Options
  - Evaluate Precondition
  - Do Not Evaluate Precondition

- **TestStand Window Activation**—Determines whether the TestStand application activates its window when the step completes. The ring control contains the following options:
  - No Activation
  - Activate When Step Completes
  - If Initially Active, Reactivate When Step Completes
- **Record Results**—Determines whether the contents of the `Result` property for the step are added to the result list for the sequence. Refer to the *Result Collection* section in Chapter 6, *Sequence Execution*, in the *TestStand User Manual* for more information on result collection.
- **Breakpoint**—Causes TestStand to break at this step before executing it. You also can set the breakpoint state for a step by selecting the **Toggle Breakpoint** item in the context menu or by clicking to the left of the step icon in the sequence editor.
- **Step Failure Causes Sequence Failure**—TestStand maintains an internal status value for each executing sequence. When TestStand sets the status property of a step to `Failed`, and you have enabled the Step Failure Causes Sequence Failure option for the step, TestStand sets the internal sequence status value to `Failed`. If the internal status of the sequence is `Failed` when the sequence returns, TestStand sets the status of the calling step to `Failed`. This affects steps that use the Sequence Call step type or the Action step type when the adapter is the Sequence adapter. Steps that use the Pass/Fail Test, Numeric Limit Test, and String Value Test step types with the Sequence adapter overwrite the step status.
- **Ignore Run-time Errors**—Prevents the step from reporting a run-time error to the sequence. When a step causes a run-time error, the step stops executing, and TestStand sets the status of the step to `Error`. If you disable this option, TestStand also sets the internal status of the sequence to `Error`, and execution branches to the Cleanup step group for the sequence. If you enable this option, TestStand does not set the internal status of the sequence to `Error`. Instead, TestStand resets the `Error.Occurred` property of the step to `False` and execution continues normally with the next step. The value of the `Result.Status` property remains set to `Error` for the step.

5. Click on the Post Actions tab, shown in Figure 3-7. The Post Actions tab specifies an action that occurs after the step executes.



On Pass: Goto next step

On Fail: Goto next step

☐ Specify Custom Condition

Custom Condition Expression:

On Condition True: Goto next step

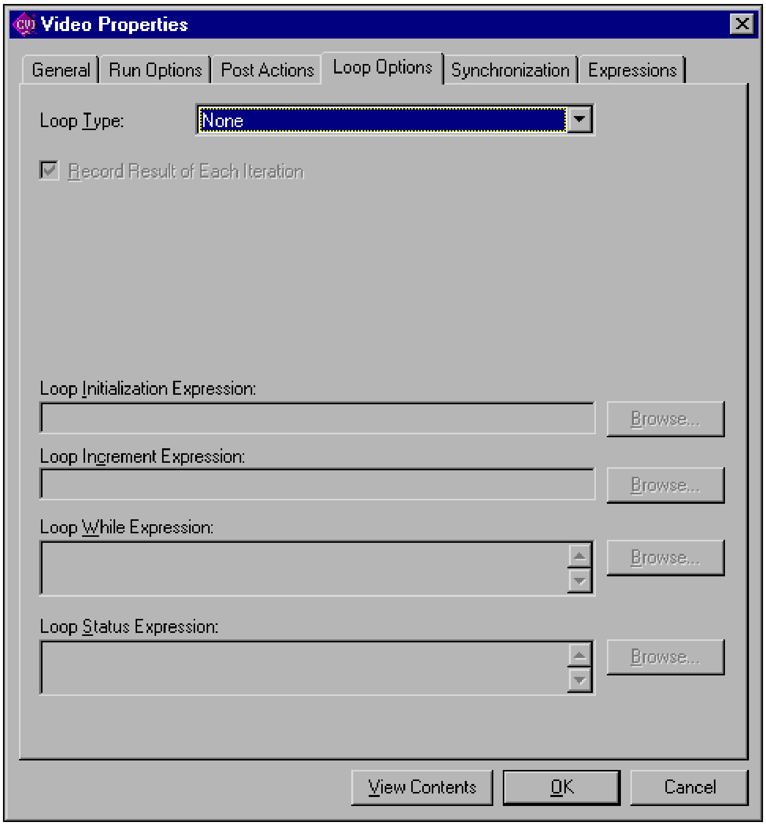
On Condition False: Goto next step

Figure 3-7. Post Actions Tab

You can make the action conditional on the pass/fail status of the step or on any custom condition expression. For example, you might want to go to a particular step or call a callback sequence if the step fails. By default the On Pass and On Fail actions are to go to the next step.

6. Set the On Fail post action to Terminate execution, which forces the sequence execution to terminate if the step fails.

7. Click on the Loop Options tab, shown in Figure 3-8.



Loop Type: None

Record Result of Each Iteration

Loop Initialization Expression:

Loop Increment Expression:

Loop While Expression:

Loop Status Expression:

Figure 3-8. Loop Options Tab

You can use the Loop Options tab to configure an individual step to run repeatedly in a loop when it executes. Use the Loop Type ring control to enter the type of looping for the step. Your choices include:

- **None**—TestStand does not loop on the step. This is the default value.
- **Fixed number of loops**—TestStand loops on the step a specific number of times and determines the final pass or fail status of the step based on the percentage of loop iterations in which the step status is `Passed`.
- **Pass/Fail count**—TestStand loops on the step until the step passes or fails a specific number of times or until a maximum number of loop iterations complete. TestStand determines the final status of

the step based on whether the specific number of passes or failures occur, or the number of loop iterations reaches the maximum.

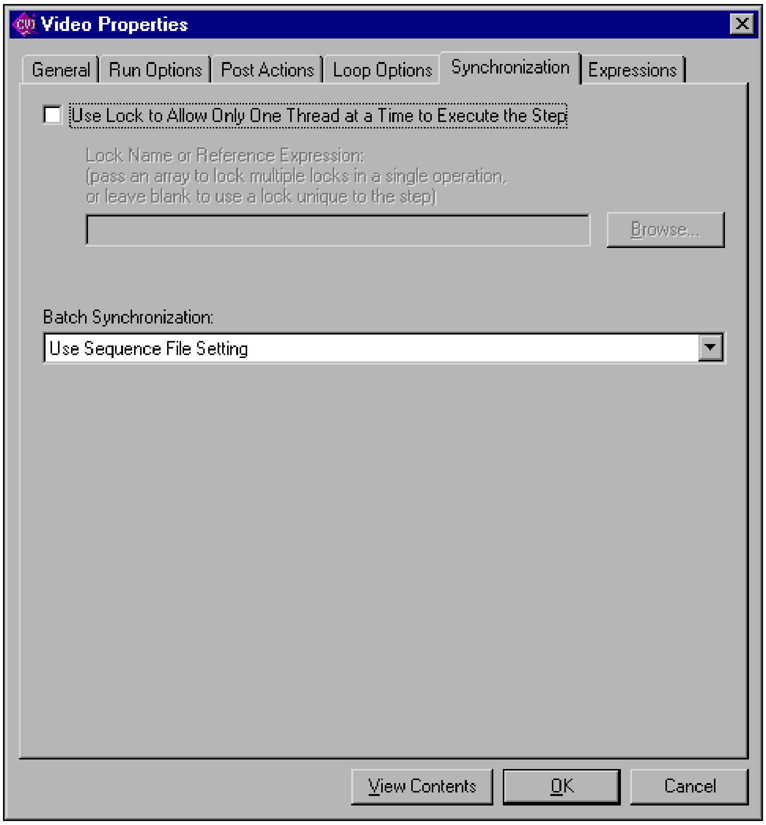
- **Custom**—This value allows you to customize the looping behavior for the step. You enter a Loop Initialization expression, a Loop Increment expression, a Loop While expression, and a final Loop Status expression.

8. Change the following control values in the Loop Options tab:

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

With these settings, TestStand executes the Video Test step ten times and sets the overall status for the step to `Failed` if less than eight of the ten iterations pass. Figure 3-8 shows the completed dialog box.

9. Click on the Synchronization tab, shown in Figure 3-9.



Use Lock to Allow Only One Thread at a Time to Execute the Step

Lock Name or Reference Expression:
(pass an array to lock multiple locks in a single operation, or leave blank to use a lock unique to the step)

Batch Synchronization:
Use Sequence File Setting

Figure 3-9. Synchronization Tab

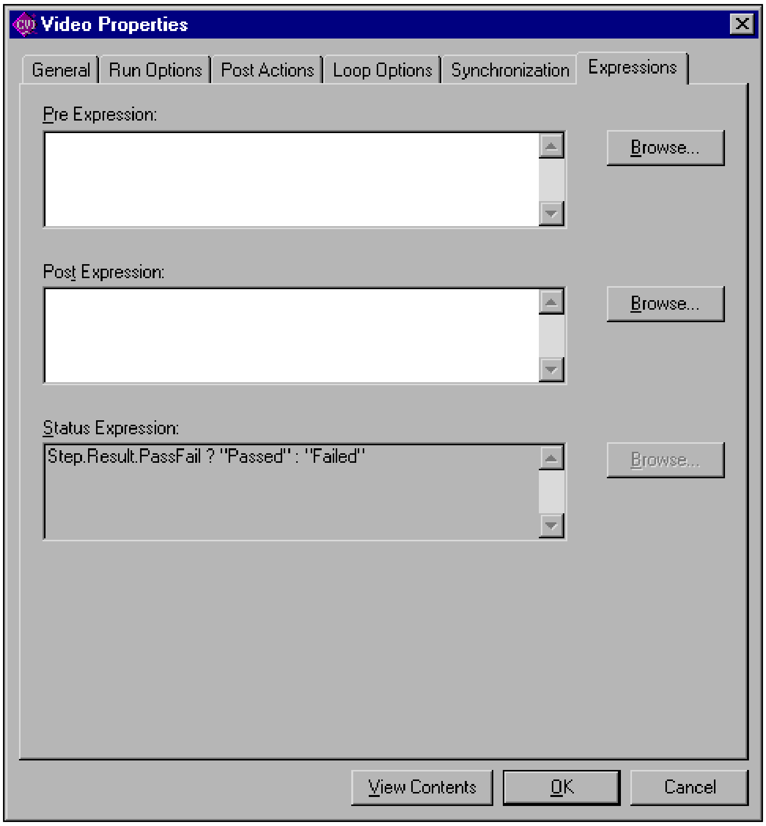
The Synchronization tab in the Step Properties dialog box specifies a synchronization action that TestStand performs around the execution of the step. You can specify that a lock protects the execution of the step or that the step synchronizes with other steps in a batch execution. For more information on synchronization step types, refer to the *Batch Synchronization* section of Chapter 11, *Synchronization Step Types,* in the *TestStand User Manual*.

- **Use Lock to Allow Only One Thread to Execute the Step**—Specifies that the step acquires a lock before it executes and releases the lock after it completes.
- **Lock Name or Reference Expression**—Specifies which lock the step acquires and releases. Enter a string expression to specify the name of an existing lock. You also can enter an expression that

evaluates to an ActiveX reference to an existing lock object. Leave the control empty to specify that TestStand uses a lock that is unique to the step.

- **Batch Synchronization**—Specifies the batch synchronization operation that the step enters before it executes and exits after it completes.

10. Click on the Expressions tab, shown in Figure 3-10.



Step.Result.PassFail ? "Passed" : "Failed"

Figure 3-10. Expressions Tab

You can use the Expressions tab to enter expressions that TestStand evaluates before and after TestStand calls the step. You learn about expressions in TestStand in Chapter 5, *Using Variables and Properties*, in this manual.

11. Click on the **OK** button to close the Step Properties dialog box.

    Notice that the Execution Flow column on the Sequence File window shows that the Video Test contains Post Actions and Loop Options.
12. Select **File»Save As** and save the sequence in the `TestStand\Tutorial` directory as `Sample2.seq`.
13. Run the sequence by selecting **Execute»Single Pass**. Notice that if you select the Video Test to fail, the sequence immediately terminates after calling the Video Test ten times in a loop.

    After TestStand generates the report, notice that when the video step executes, the result of each loop iteration is recorded in the report.
14. Close the Execution window by selecting **File»Close** or by clicking on the X icon on the window title bar.

Calling a Subsequence from a Sequence

In TestStand, you can call another sequence using a Sequence Call step in a calling sequence. You can locate your called sequence within the calling sequence file or a separate sequence file. In this exercise, you add a sequence call step to your current sequence.

1. Right click on the Power On step and select the **Insert Step» Sequence Call** command in the context menu. When you make this selection, the sequence editor inserts a Sequence Call step after the Power On step.
2. Rename the step `CPU Test`.
3. Specify which sequence the step invokes, as follows:
    a. Right click on the CPU Test step.
    b. Select the **Specify Module** command from the context menu, which displays the Edit Sequence Call dialog box.
    c. Click on the **Browse** button to the right of the File Pathname control.

d. Select the file `SubSequence1.seq` from the `TestStand\Tutorial` directory. Figure 3-11 shows the completed dialog box.

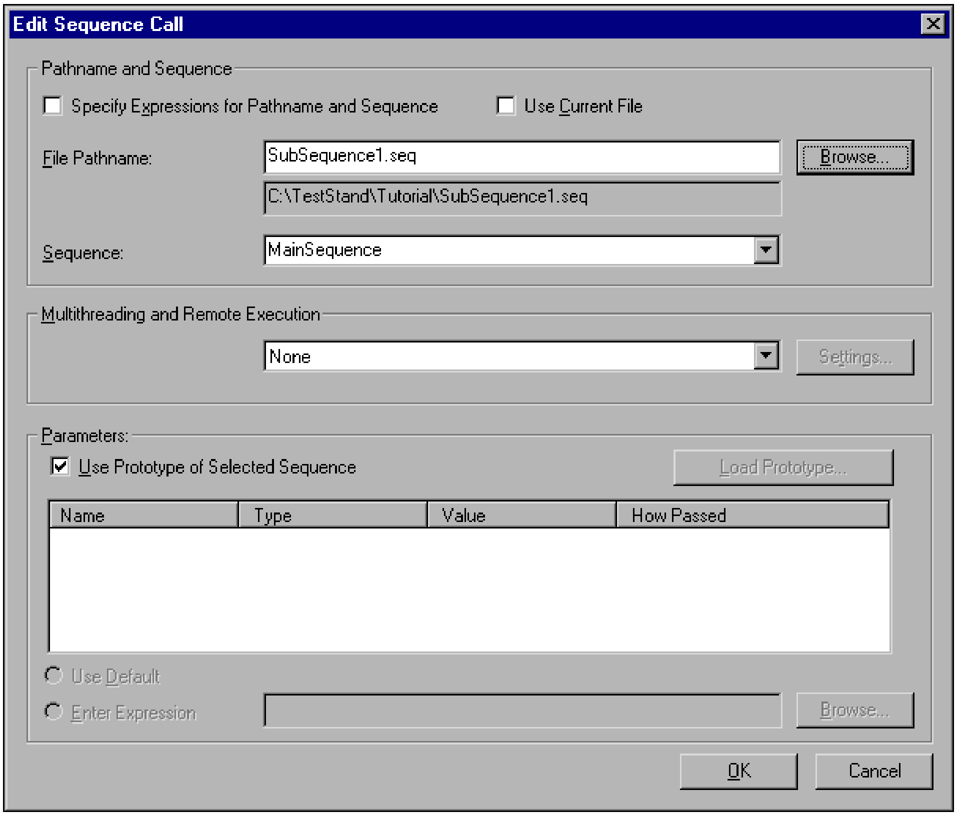


Figure 3-11. Edit Sequence Call Dialog Box

e. Click on the **OK** button to close the dialog box.

4. Select **File»Save** to save your changes to the sequence file.

5. Right click on the CPU Test step again and select the **Open Sequence** command from the context menu. When you make this selection, the sequence editor opens the `SubSequence1.seq` sequence file and displays the `MainSequence` sequence.

   As with all sequence files, you can execute any sequence in the sequence file.

6. Select **Execute»Run MainSequence**. Examine the execution of this sequence.
7. Close the Execution window.
8. Close the `SubSequence1.seq` Sequence File window.
9. Select **Execute»Single Pass**.
10. Select a test to fail.
11. Click on **Done**.

    After the sequence executes, examine the test report and notice that TestStand logs the results of the steps in the subsequence along with the steps from the parent sequence.
12. Close the Execution window.

The sequence call step provides alternate methods of calling subsequences. The Multithreading and Remote Execution section of the Edit Sequence Call dialog box allows you to call a subsequence on a thread parallel to that of the calling sequence. You can use the ring control to specify that the sequence you call runs in a new thread within the current execution or in a new execution. In addition, the ring control allows you to invoke a subsequence in a TestStand engine that runs on a remote host as a server. The Settings button displays a dialog box that is unique for each selection in the ring control. Use this dialog box to further configure your sequence call preferences. Refer to Chapter 13, *Module Adapters,* in the *TestStand User Manual* for more information about calling sequences.

This concludes this tutorial session. In the next session, you learn how to use the executing and debugging tools available in TestStand.

4

Debugging Sequences

In this chapter, you use the sequence debugging features of TestStand to single-step through your sequence during an execution. You debug the source code in a test in Chapter 6, *Creating and Debugging Tests*.

Setting Up the Example

If you did not directly proceed from Chapter 3, *Editing Steps in a Sequence*, follow these steps to set up the TestStand sequence editor so you can complete this tutorial session:

1. Close all windows in the sequence editor.
2. Select **File»Open** and open the file `<TestStand>\Tutorial\Sample2.seq`, which you created in Chapter 3, *Editing Steps in a Sequence*. You also can find this file in the `<TestStand>\Tutorial\Solution` directory.
3. Display the `MainSequence` sequence in the Sequence File window by selecting `MainSequence` in the View ring.

Step Mode Execution

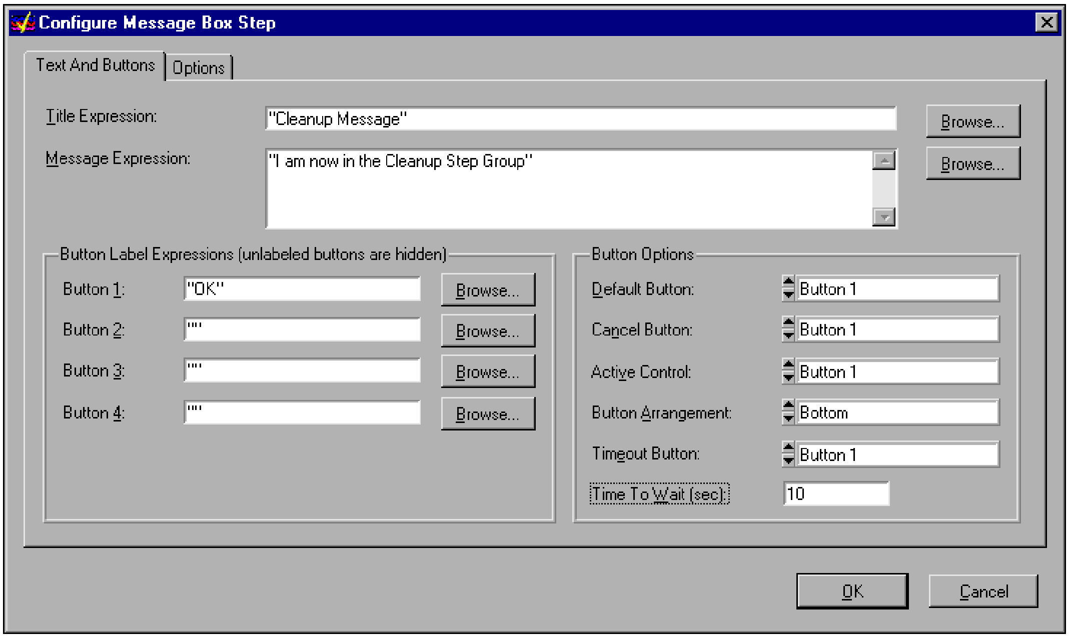
To try step mode execution in TestStand, follow these instructions:

1. Select **Execute»Break At First Step**. You use this command to suspend an execution on the first step that TestStand executes. When enabled, this command has a checkmark beside it in the menu.
2. Click on the Cleanup tab in the sequence window.

   Recall that TestStand executes the Cleanup step group after the Main step group executes, regardless of whether the sequence completes successfully or a run-time error occurs in the sequence. If a Setup or Main step causes a run-time error to occur, the flow of execution stops and jumps to the Cleanup step group.

3. Add a step that displays a message popup to the Cleanup step group to verify this behavior, as follows:
   a. Right click in the empty step list of the Cleanup tab and select **Insert Step»Message Popup** from the context menu.
   b. Rename the new step `Cleanup Message.`
   c. Right click on the Cleanup Message step and select the **Edit Message Settings** command from the context menu. When you make this selection, the sequence editor displays the Configure Message Box Step dialog box.
   d. Enter the text `"Cleanup Message"` into the Title Expression string control. Ensure that you enclose the string with double quotation marks (`"`), which indicates to TestStand that the expression is a literal string.
   e. Enter the text `"I am now in the Cleanup Step Group"`, including the quotation marks, into the Message Expression string control.
   f. Select `Button 1` from the Timeout Button control. This will enable the Time to Wait control. The Timeout Button selection ring specifies which message box button activates automatically after a timeout period expires.
   g. Enter the value of `10` into the Time to Wait control. When your sequence is executed, your message popup step displays a notification message that automatically dismisses itself if the user does not make an entry within 10 seconds. This is useful if an operator is not present to acknowledge a non-critical message that displays during testing.

      Figure 4-1 shows the Text and Buttons tab of the completed dialog box.


Configure Message Box Step

Text And Buttons | Options

Title Expression: "Cleanup Message"

Message Expression: "I am now in the Cleanup Step Group"

Button Label Expressions (unlabeled buttons are hidden)

Button 1:	"OK"
Button 2:	""
Button 3:	""
Button 4:	""

Button Options

Default Button:	Button 1
Cancel Button:	Button 1
Active Control:	Button 1
Button Arrangement:	Bottom
Timeout Button:	Button 1
Time To Wait (sec):	10

Figure 4-1. Configure Message Box Step Dialog Box

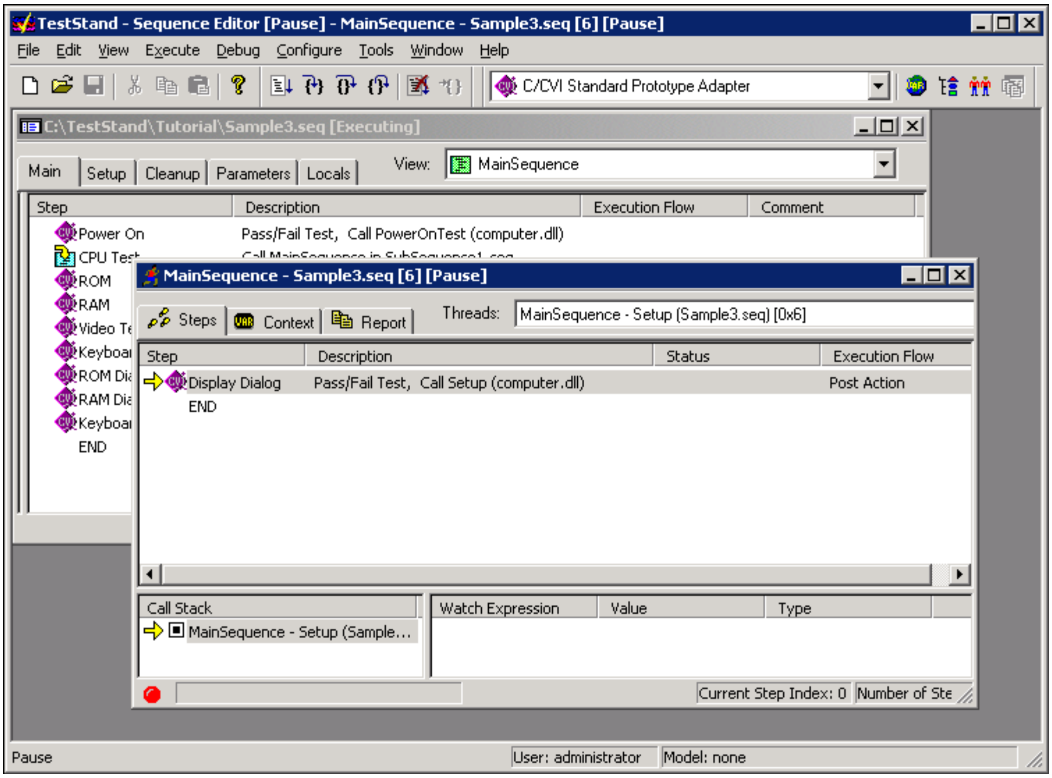
The Options tab of the Configure Message Box Step dialog box allows you to enable a response text box for operator input, position the message popup using coordinates, and make the message popup modal with respect to the application. A modal dialog box is a dialog box that you must dismiss before you can operate other application windows.

h. Click on the **OK** button to close the dialog box.

4. Save the sequence by selecting **File»Save As**. Save the sequence as `Sample3.seq` in the `TestStand\Tutorial` directory.

5. Execute the sequence directly by selecting **Execute»Run MainSequence**.

   After the execution starts, the sequence editor immediately pauses the execution on the first step of the sequence because you previously enabled the Break On First Step option. Figure 4-2 shows the current state of the sequence editor.


TestStand - Sequence Editor [Pause] - MainSequence - Sample3.seq [6] [Pause]
File Edit View Execute Debug Configure Tools Window Help
C/CVI Standard Prototype Adapter
C:\TestStand\Tutorial\Sample3.seq [Executing]
Main Setup Cleanup Parameters Locals
View: MainSequence
MainSequence - Sample3.seq [6] [Pause]
Steps Context Report
Pause

Threads: MainSequence - Setup (Sample3.seq) [0x6]

Step	Description	Execution Flow	Comment
Power On	Pass/Fail Test, Call PowerOnTest (computer.dll)		
CPU Test	Call MainSequence in SubSequence1.seq		
ROM			
RAM			
Video Te			
Keyboar			
ROM Dia			
RAM Dia			
Keyboar			
END			

Step	Description	Status	Execution Flow
Display Dialog	Pass/Fail Test, Call Setup (computer.dll)		Post Action
END			

Call Stack
MainSequence - Setup (Sample...

Watch Expression	Value	Type

Current Step Index: 0
Number of Ste

User: administrator
Model: none

Figure 4-2. Paused Execution of Sample3.seq

Notice that the title of the Execution window contains the running state of the execution, that is, [Pause]. When execution suspends, the Steps tab in the Execution window displays the execution pointer next to the step that will run when execution resumes. The next step that TestStand will execute is the Display Dialog step in the Setup step group.

When execution is in the paused state, you can single-step through the sequence using the **Step Into**, **Step Over**, and **Step Out** commands in the **Debug** menu, or by using the toolbar buttons, as shown in Figure 4-3. For a detailed discussion of the single-stepping tools, refer to Chapter 6, *Sequence Execution*, in the *TestStand User Manual*.

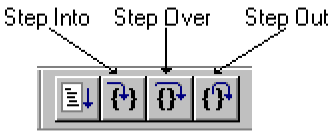
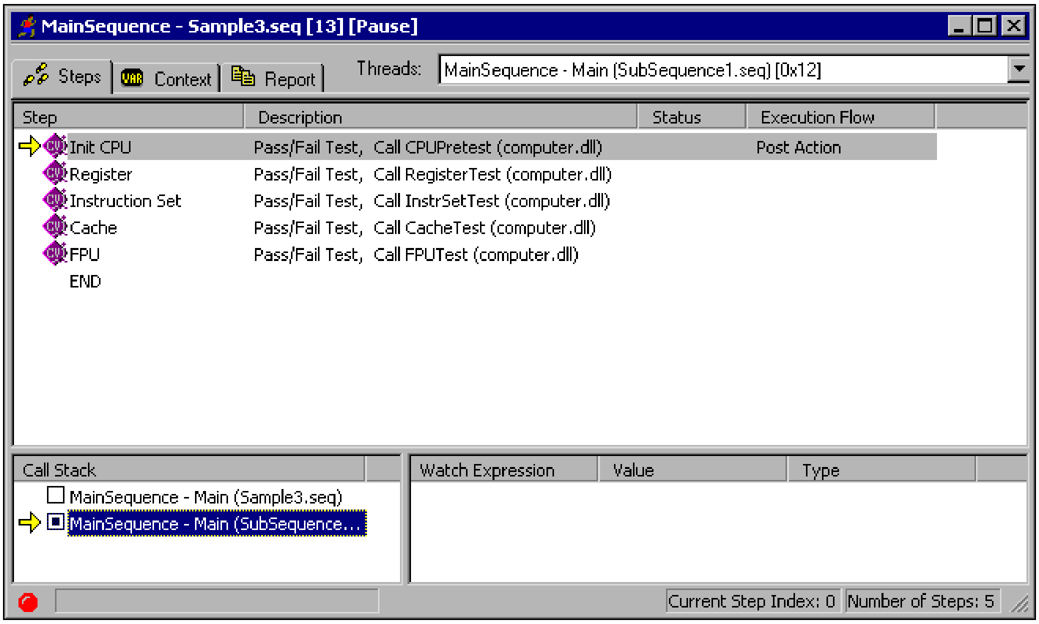


Figure 4-3. Single-Stepping Toolbar Buttons

6. Click on the **Step Over** toolbar button to execute the Display Dialog step. This step displays the Test Simulator dialog box.
7. Select the RAM test to fail, and click on **Done**.

   After you close the dialog box, the sequence editor suspends the sequence execution at the end of the Setup step group on END.
8. Activate the `Sample3.seq` Sequence File window by clicking on it or by selecting **Window»Sample3.seq**.
9. Right click on the CPU Test step in the Main step group tab, and select the **Toggle Breakpoint** command in context menu. Notice that a red stop sign icon appears to the left of the step name.
10. Return to the Execution window by clicking on it or by selecting the window in the **Window** menu. Select **Debug»Resume** to continue the execution. After you make this selection, the sequence editor suspends the execution again on the CPU Test step.
11. Click on the **Step Into** toolbar button to step into the subsequence.

Figure 4-4 shows the Steps view for the Execution window after you step into the subsequence.



MainSequence - Main (SubSequence1.seq) [0x12]

Step	Description	Status	Execution Flow
Init CPU	Pass/Fail Test, Call CPUPretest (computer.dll)		Post Action
Register	Pass/Fail Test, Call RegisterTest (computer.dll)		
Instruction Set	Pass/Fail Test, Call InstrSetTest (computer.dll)		
Cache	Pass/Fail Test, Call CacheTest (computer.dll)		
FPU	Pass/Fail Test, Call FPUTest (computer.dll)		
END			

Call Stack
MainSequence - Main (Sample3.seq)
MainSequence - Main (SubSequence...

Watch Expression	Value	Type

Current Step Index: 0 Number of Steps: 5

Figure 4-4. Steps View while Suspended in Subsequence

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

The Call Stack pane in the lower left half of the Execution window displays the call stack for the execution. A yellow pointer icon appears to the left of the most nested sequence invocation. The call stack in Figure 4-4 shows that the main sequence in `Sample3.seq` is calling the main sequence in `SubSequence1.seq`.

When execution suspends, you can view a sequence invocation in the call stack by clicking on its radio button.

12. Click on each radio button in the Call Stack pane to view the status of each sequence invocation.
13. Return to the bottom of the most nested sequence invocation in the call stack.

14. Click on the **Step Over** toolbar button twice to start stepping through the subsequence.
15. Before you reach the end of the sequence, select the **Step Out** toolbar button. TestStand resumes the execution through the end of the current sequence and suspends the execution before the next step after the sequence call or until it reaches a breakpoint, which ever comes first.
16. Continue single-stepping through the sequence using the **Step Over** toolbar button until the execution completes. Notice that the last step executed is the Cleanup Message step that you added to the Cleanup step group.
17. Click on **OK** to close the cleanup message before you complete the execution. The Execution window dims when the execution completes. Do not close the Execution window.
18. Rerun the execution by selecting **Execute»Restart**. The Execution window must be the active window to restart the sequence.
19. After the sequence editor suspends the execution on the first step, select **Debug»Terminate**.

    Notice that TestStand still displays the Cleanup Message dialog box even though you terminated the sequence execution. An execution proceeds immediately to the steps in the Cleanup step group when an operator or a run-time error terminates the execution.
20. Click on **OK** to close the cleanup message.
21. Rerun the execution again by selecting **Execute»Restart**.
22. After the sequence editor suspends the execution on the first step, select **Debug»Abort**. Notice that the execution of the sequence immediately stops, and TestStand does not execute any steps in the Cleanup step group.
23. Close the Execution window.
24. Save the sequence by selecting **File»Save**.
25. Close the `Sample3.seq` window.

This concludes this tutorial session. In the next session, you learn how to create and use TestStand variables and properties.

5

Using Variables and Properties

This chapter teaches you how you can use variables and properties in TestStand, and points out features of TestStand that help you monitor the values of variables and properties.

In TestStand, you can define variables with various scopes to share data between steps of a sequence or even between several sequences. You can define variables that are local to a sequence, variables that are global to a sequence file, and variables that are global to the test station. Use these types of variables as follows:

- You can use *local variables* to store data relevant to the execution of the sequence. Each step and step module can directly access sequence local variables.
- You can use *sequence file global* variables to store data relevant to the entire sequence file. Each sequence and step in the sequence file can directly access these globals.
- You can access *station global* variables from any sequence, step or code module. Unlike other variables, the values of station global variables are saved from one TestStand session to the next. Normally, you use station global variables to maintain statistics or to represent the configuration of your test station.

Setting Up the Example

If you did not directly proceed from Chapter 4, *Debugging Sequences*, close all windows in the sequence editor so you can complete this tutorial session.

Using TestStand Variables

In this exercise, you learn how to create and use local variables. You can apply the concepts that you learn to sequence file globals and station globals.

1. Select **File»Open** and open the file `<TestStand>\Tutorial\Sample2.seq`, which you created in Chapter 3, *Editing Steps in a*

Sequence. You also can find this file in the `<TestStand>\Tutorial\Solution` directory.

2. Display the `MainSequence` sequence in the sequence file window by selecting `MainSequence` in the View ring.
3. Click on the Locals tab of the sequence window. When you make this selection, the view displays all of the local variables currently defined for `MainSequence` in the `Sample2.seq` sequence file. By default, TestStand defines only one local variable, `ResultList`, when creating a new sequence. TestStand uses this array variable to store the results from the steps it executes in this sequence. This array of step results is used in report generation.
4. Right click in the right pane and select **Insert Local»Number** from the context menu, as shown in Figure 5-1. When you make this selection, the sequence editor inserts a new numeric local variable.

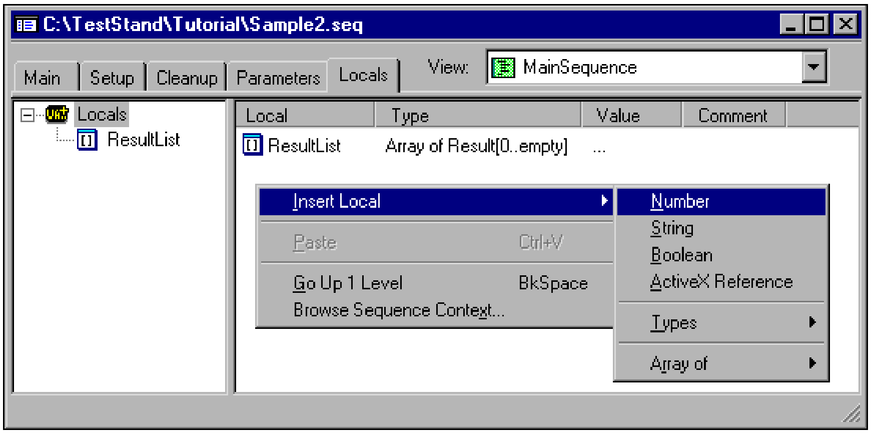


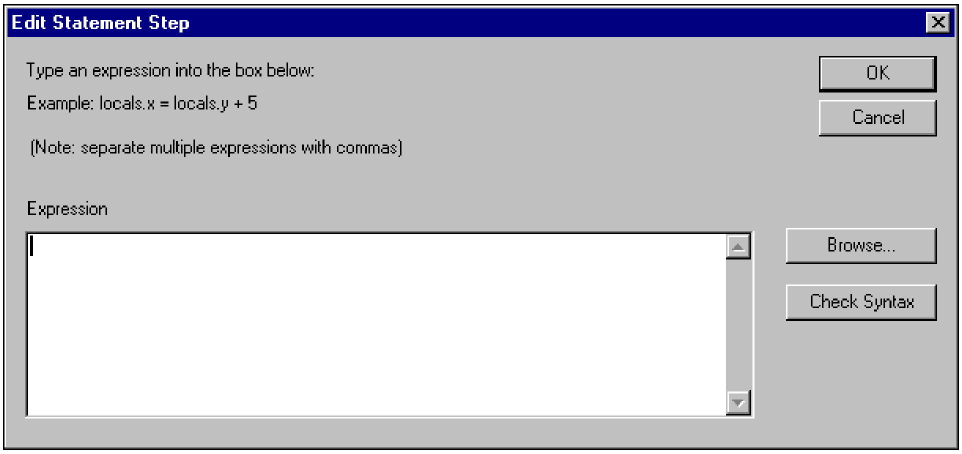
Figure 5-1. Insert Local Context Menu Command

5. Rename the variable `LoopIndex`.

Note The name of a TestStand variable cannot begin with a number or contain any spaces.

6. Add steps to the sequence to make it loop on a set of steps based on the value of the `LoopIndex` local variable, as follows:
   a. Click on the Main tab in the sequence file window to display the steps in the Main step group.
   b. Right click on the Power On test and select the **Insert Step» Statement** command from the context menu.
   c. Rename the new step `Reset Loop Index`.

      You use Statement steps to execute expressions that TestStand evaluates when it executes the step. For example, you can use a Statement step to increment the value of a local variable in the sequence file.
   d. Right click on the Reset Loop Index step and select the **Edit Expression** command from the context menu, which displays the Edit Statement Step dialog box, shown in Figure 5-2.



Type an expression into the box below:
Example: locals.x = locals.y + 5
(Note: separate multiple expressions with commas)

Figure 5-2. Edit Statement Step Dialog Box

e. Click on the **Browse** button to display the Expression Browser dialog box, shown in Figure 5-3.

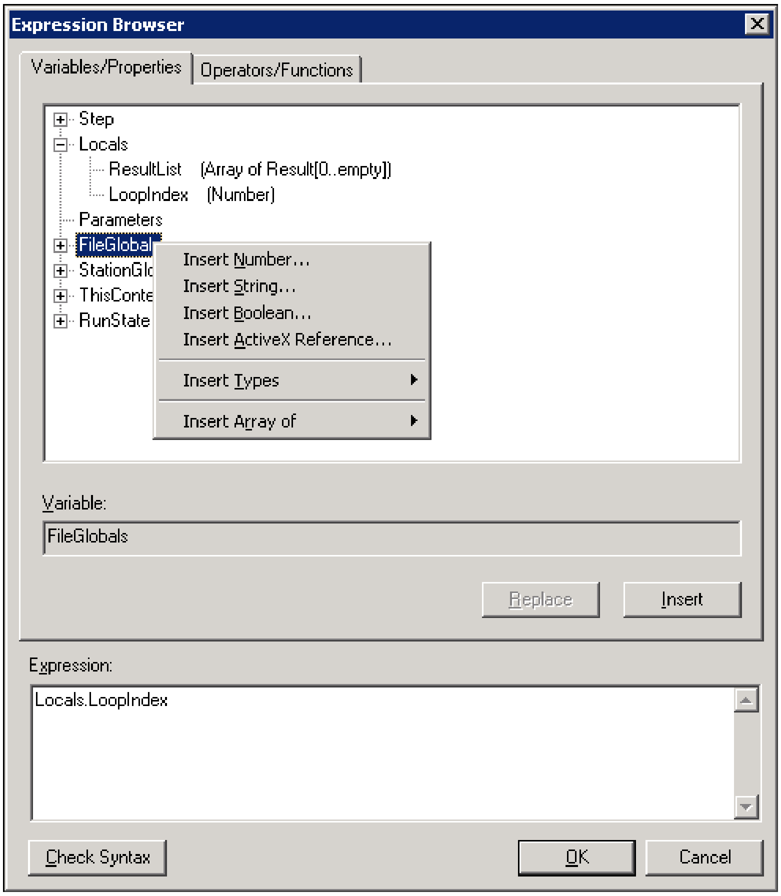


Figure 5-3. Expression Browser Dialog Box

You use the Expression Browser to interactively build an expression and to create variables and parameters. The Expression Browser contains two tabs, Variables/Properties and Operators/Functions. You can select variables and properties from the tree view on the Variables/Properties tab. The Operators/Functions tab contains a list of all predefined operators and functions.

The expression browser has help text for the currently selected operator or function. TestStand supports all applicable expression operators and syntax that you use in C, C++, Java, and Visual Basic.

You can create variables directly from the Expression Browser using the context menu shown in Figure 5-3. Hold your mouse pointer over properties shown in the Expression Browser to get tip strips. The tip strip displays **Right-click to insert new variable** for those properties under which you can create a variable.

f. Expand the Locals item by double clicking on the name or by clicking on the plus icon in front of the item. When you expand a tree view item, the dialog box displays all the items under the base item. Each item in the tree view is a property or a variable of TestStand.

g. Select the `LoopIndex` variable under the Locals property and click on the **Insert** button. When you make this selection, the expression browser enters `Locals.LoopIndex` into the Expression control.

   To refer to a subproperty, you use a period to separate the name of the property from the name of the subproperty. For example, you reference the `LoopIndex` subproperty in the `Locals` property as `Locals.LoopIndex`.

h. Click on the Operators/Functions tab, and select the Assignment category from the left pane.

i. Select the assignment operator (`=`) from the right pane.

j. Click on **Insert** to add the assignment operator to the expression. You should now see `Locals.LoopIndex =` in the Expression control.

k. Place the text cursor directly after the equals sign in the expression control and then type the number zero so that the expression now reads `Locals.LoopIndex = 0`.

l. Click on the **OK** button to return to the Edit Statement Step dialog box.

m. Click on the **Check Syntax** button to verify that the expression does not contain any illegal syntax.

n. Close the Edit Statement Step dialog box by clicking on the **OK** button to return to the Sequence Editor window.

o. Right click on the Reset Loop Index step and select **Insert Step» Label** from the context menu.

   p. Rename the new step `Loop Begin`. You normally use a Label step as the target for a Goto step, as you will see later in this session. By using a Label step, you can rearrange or delete other steps around the Label step without having to change the target step that the Goto step references.

7. Add a statement step to increment the value of the `LoopIndex` local variable, as follows:
   a. Right click on the RAM test and select **Insert Step»Statement** from the context menu.
   b. Rename the new step `Increment Loop Index`.
   c. Right click on the Increment Loop Index step and select **Edit Expression** from the context menu.
   d. Click on the **Browse** button to display the expression browser.
   e. Use the expression browser to build, or type directly in the Expression control, the following expression:

      `Locals.LoopIndex ++`

      The increment operator (`++`) is under the Arithmetic group of the Operators/Functions tab.
   f. Click on **OK** twice to close the both the Expression Browser and the Edit Statement dialog boxes.
8. To complete the loop structure, add a Goto step to the sequence, as follows:
   a. Right click on the Increment Loop Index step and select **Insert Step»Goto** from the context menu.
   b. Rename the step `Loop End`.
   c. Right click on the Loop End step and select **Edit Destination** from the context menu.
   d. Select the Loop Begin step in the Destination control by clicking on the arrow to the right of the control.
   e. Click on **OK** to close the dialog box.
9. To complete the loop structure, set a precondition for the Loop End step so that it executes only if the value of the `LoopIndex` variable is below a certain value, as follows:
   a. Right click on the Loop End step and select **Properties** from the context menu.
   b. Click on the **Preconditions** button to open the Preconditions dialog box.

c. Click on **Insert New Expression**.

d. Click on the **Browse** button in the Edit/View Expression section of the dialog box to open the Expression Browser.

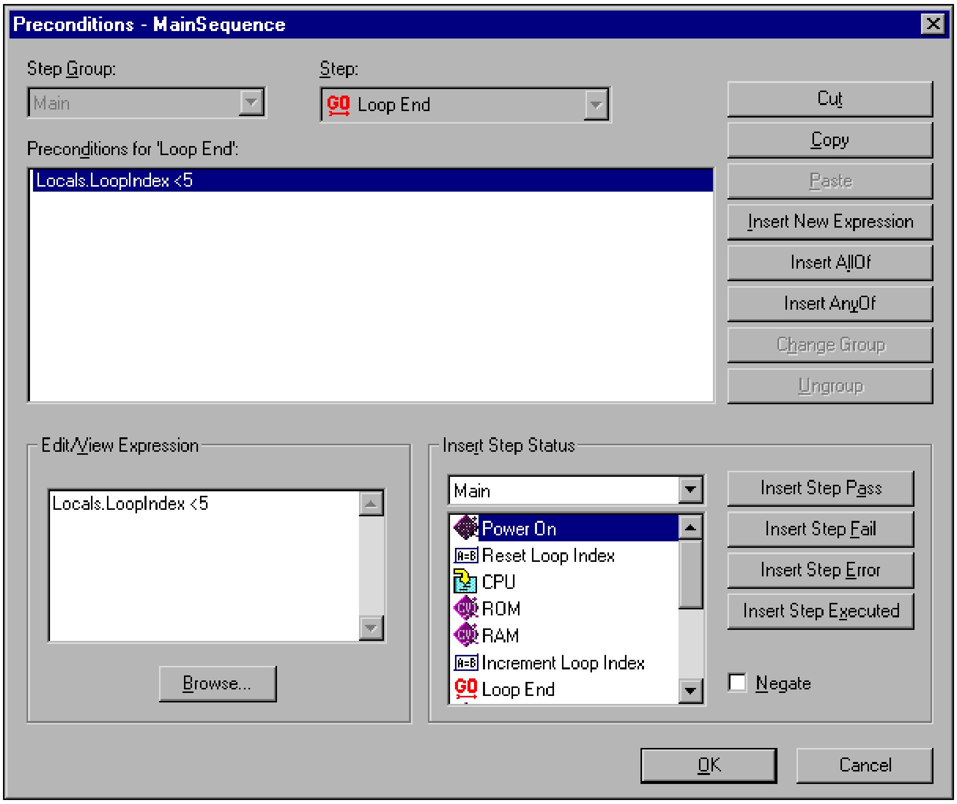
e. Using the Expression Browser, create the following expression:

```
Locals.LoopIndex < 5
```

The less than operator (<) is under the Comparison group on the Operators/Functions tab.

f. Click on **OK** to close the Expression Browser dialog box.

Figure 5-4 shows the completed Preconditions dialog box.



Preconditions for 'Loop End':

Locals.LoopIndex <5

Figure 5-4. Preconditions for the Loop End Step

10. Click on **OK** twice to close the both the Preconditions dialog box and the Step Properties dialog box.

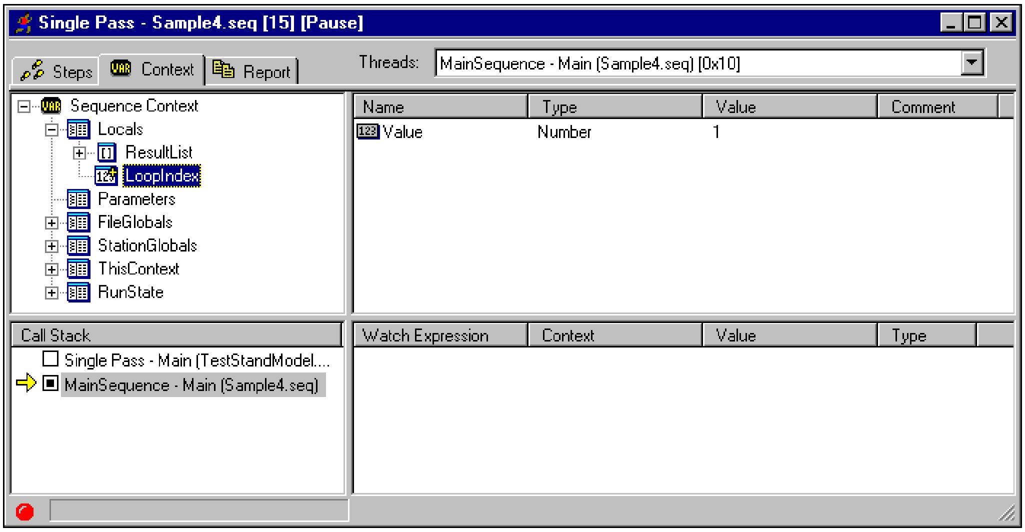
11. Select **File»Save As**. Save the sequence as `Sample4.seq` in the `TestStand\Tutorial` directory.
12. Click on the **Execute** menu to see if the **Break At First Step** option is enabled.
13. If **Break At First Step** is enabled, disable it.
14. Run the Sequence by selecting **Execute»Single Pass**.
15. Click on **Done** in the Test Simulator dialog box.
16. After the sequence executes, examine the test report and notice that TestStand executed the steps within the loop (CPU Test, ROM Test, and RAM Test) five times.
17. Close the Execution window.

Using the Context Tab

In this exercise, you use the Watch Expression pane of the Execution window to examine the value of the `LoopIndex` variable while TestStand executes the sequence.

1. Right click on the Loop End goto step and select **Toggle Breakpoint** from the context menu to set a breakpoint on the step. In the sequence window, a red stop sign icon appears beside the Loop End step indicating this breakpoint.
2. Run the sequence by selecting **Execute»Single Pass**.
3. Click on **Done** in the Test Simulator dialog box. The execution suspends on the Loop End step.
4. Click on the Context tab of the Execution window.
5. Expand the `Locals` property in the left upper tree view pane.

6. Click on the `LoopIndex` property under the `Locals` property and notice that the value of the numeric is 1, as shown in Figure 5-5.



Threads: MainSequence - Main (Sample4.seq) [0x10]

Name	Type	Value	Comment
Value	Number	1	

Figure 5-5. Context Tab

The Context tab displays the sequence context for the sequence invocation that is currently selected in the Call Stack pane. The sequence context contains all the variables and properties that the steps in the selected sequence invocation can access. You use the Context tab to examine and modify the values of these variables and properties.

Before executing the steps in a sequence, TestStand creates a run-time copy of the sequence. This allows parallel executions of the same sequence to run such that each execution does not alter variable or property values in other executions. When an execution completes, TestStand discards the run-time sequence copy.

TestStand maintains a *sequence context* for each active sequence. The sequence context represents the execution state of the sequence. The contents of the sequence context change depending on the currently executing sequence and step. Both the run-time copy and the original versions of properties and variables are accessible from a sequence context.

You can use the sequence context to access variables and step properties in expressions and through calls to the TestStand ActiveX API from step modules. Refer to the *Using the Watch Expression Pane* section in this chapter for information on expressions. For more information on the TestStand API, refer to the *TestStand Programmer Help*.

Table 5-1 lists the first-level properties in the sequence context and describes their contents. Refer to Chapter 8, *Sequence Context and Expressions*, in the *TestStand User Manual* for more information on sequence contexts.

Table 5-1. First-Level Properties of the Sequence Context

Sequence Context Subproperty	Description
`Step`	Contains the properties of the currently executing step in the current sequence invocation. The `Step` property exists only while a step executes. It does not exist when the execution is between steps, as at a breakpoint.
`Locals`	Contains the sequence local variables for the current sequence invocation.
`Parameters`	Contains the sequence parameters for the current sequence invocation.
`FileGlobals`	Contains the sequence file global variables for the current execution.
`StationGlobals`	Contains the station global variables for the engine invocation. TestStand maintains a single copy of the station globals in memory.
`ThisContext`	Holds a reference to the current sequence context. You normally use this property to pass the entire sequence context as an argument to a subsequence or a step module.
`RunState`	Contains properties that describe the state of execution in the sequence invocation, such as the current step, the current sequence, and the calling sequence.

7. Select **Debug»Resume** and notice that the execution resumes and suspends at the Loop End goto step again.
8. Click on the Context tab again and notice that the `LoopIndex` value is now 2. Leave the execution in the Pause state.

Using the Watch Expression Pane

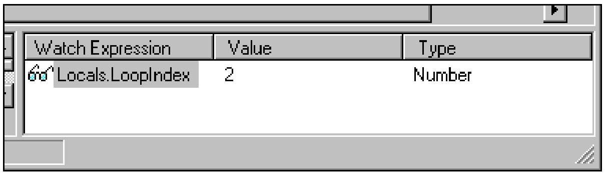
In this exercise, you monitor the value of the variable `LoopIndex` in the Watch Expression pane. The Watch Expression pane is located in the lower right of the Execution window, as shown in Figure 5-6. The Watch Expression pane displays the values of watch expressions you enter. TestStand updates the values in the Watch Expression pane when execution suspends at a breakpoint. If tracing is enabled, TestStand also updates the values after executing each step.

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

To create a watch expression for the variable `LoopIndex`, complete the following steps:

1. Click on the `LoopIndex` property in the tree view of the Context tab, and while holding down the mouse button, drag the variable from the tree view to the Watch Expression pane. Release the mouse button when the cursor is over the Watch Expression pane.

   Notice that the value of the watch expression already evaluates to 2, as shown in Figure 5-6.



Watch Expression	Value	Type
Locals.LoopIndex	2	Number

Figure 5-6. Updated Watch Window Pane

2. Now, select **Debug»Resume** and notice that when the execution suspends on the Goto step again, the value of the watch expression changes from 2 to 3.
3. Remove the breakpoint by clicking to the left of the Loop End step icon.
4. Select **Debug»Resume** to complete the execution.
5. Close the Execution window.
6. Save the sequence by selecting the **File»Save**, which saves the changes you made to the `Sample4.seq` sequence file.

You can create more complex expressions in the Watch Expression pane. To add a new expression, right-click in the Watch Expression pane and select **Add Watch** from the context menu. To edit an existing watch expression, right-click on the expression and select **Edit Expression** from the context menu. Both of these selections display the Expression Browser that you can use to create an expression.

Note You can copy watch expressions and paste them into the Watch Expression pane of a subsequent execution. Watch expressions are automatically maintained in subsequent executions if you select **Execution»Restart** after an execution has completed but before you close the execution display window.

For more information about the Watch Expression pane, refer to Chapter 6, *Sequence Execution*, in the *TestStand User Manual*. For more details on using variables and properties in TestStand, refer to Chapter 5, *Sequence Files*, Chapter 7, *Station Global Variables*, and Chapter 8, *Sequence Context and Expressions*, in the *TestStand User Manual*. You can also refer to the following *TestStand Programmer Help* topics for information about using variables and properties in TestStand: *Object Relationships*, *Sequence Context*, *Property Paths*, *Using Property Paths*, *Finding a Property Path*, *Viewing Step Properties, Commonly Used Properties.*

This concludes this session of the tutorial. In the next session, you learn how to create and debug tests in the LabVIEW and LabWindows/CVI development environments.

6 Creating and Debugging Tests

In this chapter, you learn how to create and debug code modules that are called using the LabVIEW Standard Prototype Adapter, the C/CVI Standard Prototype Adapter, and the DLL Flexible Prototype Adapter. You use LabVIEW and LabWindows/CVI to write and debug code modules.

If you do not use LabVIEW or LabWindows/CVI, you may want to skip the exercises that use the LabVIEW Standard Prototype Adapter and the C/CVI Standard Prototype Adapter. If your code modules are C-style DLLs, the exercises using the DLL Flexible Prototype Adapter might be useful, regardless of your application development environment.

You can create and debug code modules written in other application development environments (ADEs). However, this is beyond the scope of this manual. Refer to National Instruments Developers Zone (NIDZ) for information about writing and debugging code using other ADEs.

Debugging a LabVIEW VI Using the LabVIEW Standard Prototype Adapter

In this exercise, you learn how to create LabVIEW test modules that you can use with TestStand, and how to debug them by stepping into a virtual instrument (VI) from the TestStand sequence editor. This session of the tutorial assumes a general familiarity with the LabVIEW development environment. If you are not using LabVIEW, but you do use LabWindows/CVI or you create C-style DLLs, you can skip this section and proceed to the *Debugging a LabWindows/CVI DLL Using the C/CVI Standard Prototype Adapter* section in this chapter.

Note Verify that you are using the appropriate version of LabVIEW with TestStand. Refer to the `readme.txt` file in the `TestStand\Doc` directory for more details.

Setting Up the Example

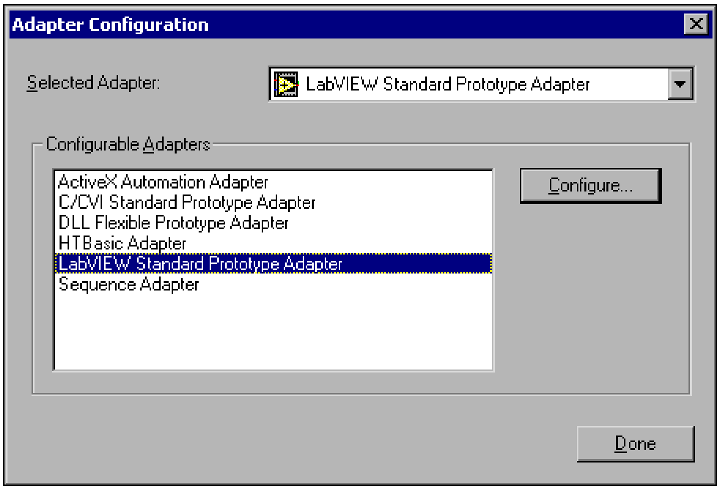
If you did not directly proceed from Chapter 5, *Using Variables and Properties*, follow these steps to set up the TestStand sequence editor so you can complete this tutorial session.

1. Close all windows in the sequence editor.
2. Select **File»Open** and open the file `<TestStand>\Tutorial\Sample4.seq` that you created in Chapter 5, *Using Variables and Properties*. You can also find this file in the `<TestStand>\Tutorial\Solution` directory.

Creating a Virtual Instrument Code Module

In this exercise, you create a LabVIEW VI you will call from a sequence in TestStand.

1. Ensure the LabVIEW Standard Prototype Adapter is properly configured, as follows:
   a. Select **Configure»Adapters**, which displays the Adapter Configuration dialog box, shown in Figure 6-1.


Adapter Configuration

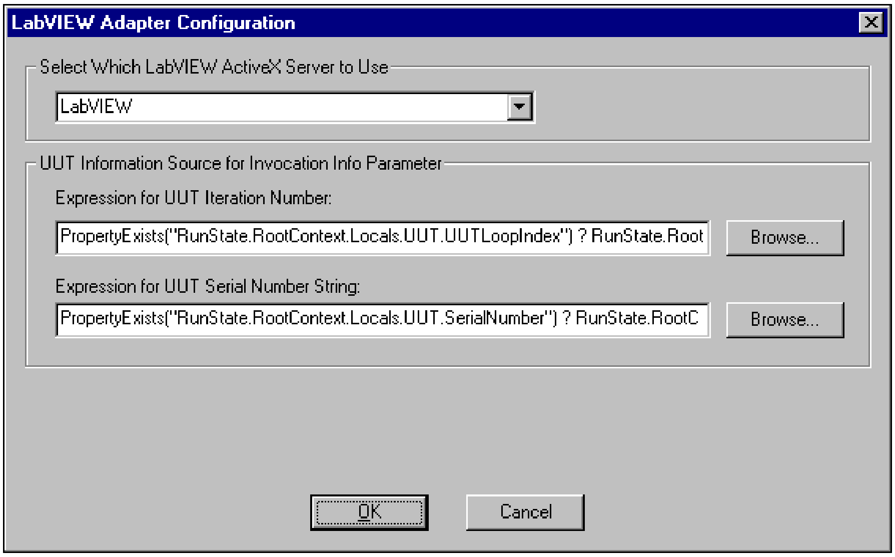
Selected Adapter: LabVIEW Standard Prototype Adapter

Configurable Adapters
- ActiveX Automation Adapter
- C/CVI Standard Prototype Adapter
- DLL Flexible Prototype Adapter
- HTBasic Adapter
- LabVIEW Standard Prototype Adapter
- Sequence Adapter

Figure 6-1. Adapter Configuration

   b. Select the LabVIEW Standard Prototype Adapter in the Configurable Adapters section.
   c. Click on the **Configure** button.

d. Verify that the Select Which LabVIEW ActiveX Server to Use control is LabVIEW, as shown in Figure 6-2.



Select Which LabVIEW ActiveX Server to Use: LabVIEW

UUT Information Source for Invocation Info Parameter

Expression for UUT Iteration Number:
PropertyExists("RunState.RootContext.Locals.UUT.UUTLoopIndex") ? RunState.Root

Expression for UUT Serial Number String:
PropertyExists("RunState.RootContext.Locals.UUT.SerialNumber") ? RunState.RootC

Figure 6-2. LabVIEW Adapter Configuration

e. Click on **OK**, and then **Done** to close the LabVIEW Adapter Configuration and Adapter Configuration dialog boxes.

2. Select LabVIEW Standard Prototype Adapter in the Adapter Selector Ring control.
3. Right click on the `Power On` test in the Main step group and select **Insert Step»Test»Numeric Limit Test** from the context menu.
4. Rename the new step `Clock Frequency Test`.
5. Right click on the Clock Frequency Test and select **Specify Module** from the context menu.

6. Enable the Show VI Front Panel When Called check box control. The LabVIEW Step Module Information should now appear as shown in Figure 6-3.

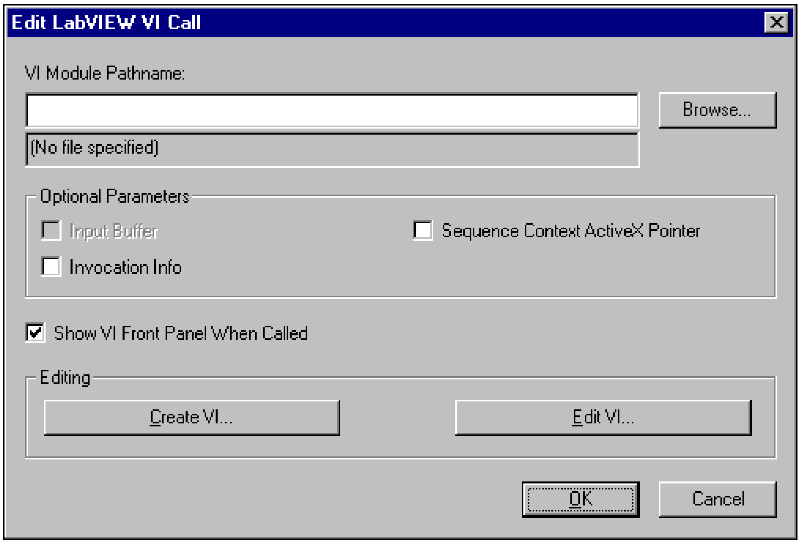


Figure 6-3. LabVIEW Step Module Information

With module adapters, you can use a source code template to generate the source code shell for a step module. The template files are different for each step type and each module adapter. Multiple source code templates can be available for a particular adapter/step type combination.

For each module adapter that supports source code templates, the Specify Module dialog box contains a button for creating source code. If more than one template is available for the step type, the adapter prompts you to select from a list of available templates. Otherwise, the adapter uses the only available template.

7. Click on the **Create VI** button on the Edit LabVIEW VI Call dialog box. When you make this selection, TestStand prompts you to select a pathname for the step's code module.
8. Find the `TestStand\Tutorial` directory. Type the name `ClockFrequency.vi` in the File name control. The VI might already exist if someone else previously completed this session of the tutorial.

9. Click on **OK** to close the Select a pathname for the step's code module dialog box.

   TestStand creates a new VI named `Clock Frequency.vi` using a code template associated with the Numeric Limit Test step type and the LabVIEW Standard Prototype adapter. TestStand then opens the new VI in LabVIEW, as shown in Figure 6-4.

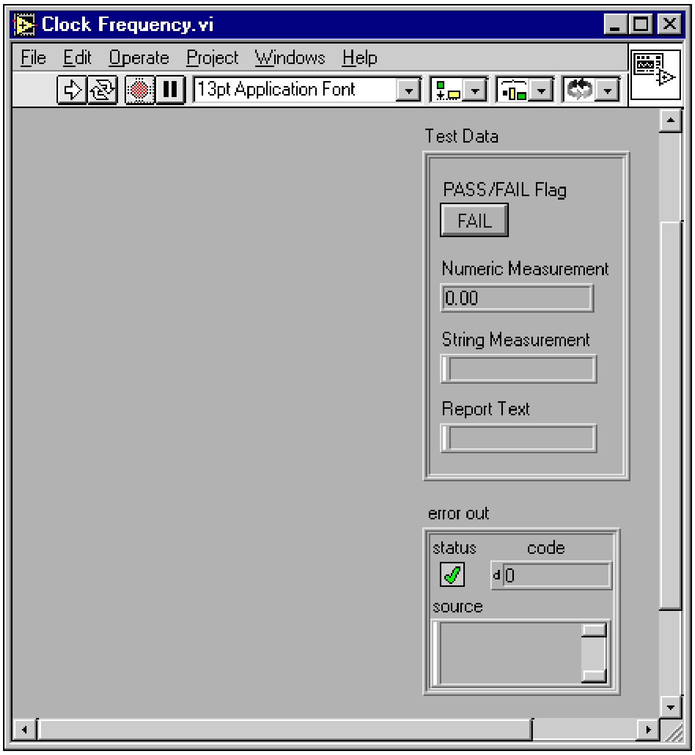


Figure 6-4. New Clock Frequency VI in LabVIEW

Notice that the `Clock Frequency.vi` front panel contains two indicators, Test Data and error out clusters. The LabVIEW Standard Prototype adapter uses these special data clusters to pass common data between TestStand and the test VI. You can use the sequence context and the TestStand API to access all sequence variables and properties and to control sequence execution, as explained in Chapter 10, *Using ActiveX in Code Modules*. Although there are other special data types that the module adapter supports for passing data, the Test Data and error out clusters are required controls. Following is a list of the different elements within these two clusters and how the adapter uses them:

Test Data

- **Pass/Fail Flag**—The test VI sets this Boolean to indicate whether the test passed.
- **Numeric Measurement**—Numeric measurement that the test VI returns.
- **String Measurement**—String value that the test function returns.
- **Report Text**—Output message to display in the report.

error out

- **Status**—The test VI must set this Boolean to `True` if an error occurs.
- **Code**—The test VI can set this to a non-zero value if an error occurs.
- **Source**—The test VI can set this to a descriptive string if an error occurs.

Refer to Chapter 13, *Module Adapters*, in the *TestStand User Manual* for more details on these structures.

10. Add the following LabVIEW controls to the front panel, as shown in Figure 6-5:
    - Numeric control with the label Frequency Measurement
    - String control with the label Additional Report Text
    - Dialog button with the label Return

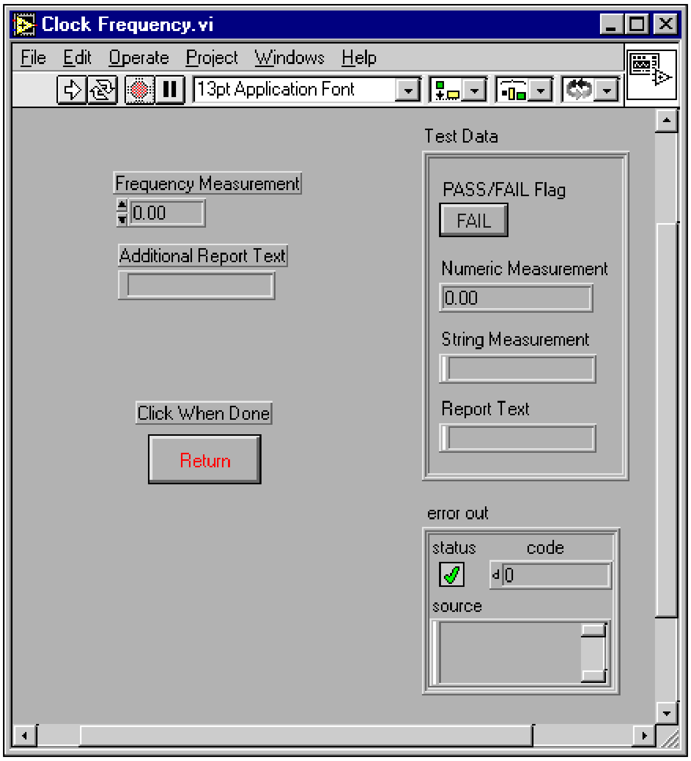


Figure 6-5. Completed Clock Frequency.vi Front Panel

11. Wire the VI block diagram as shown in Figure 6-6.

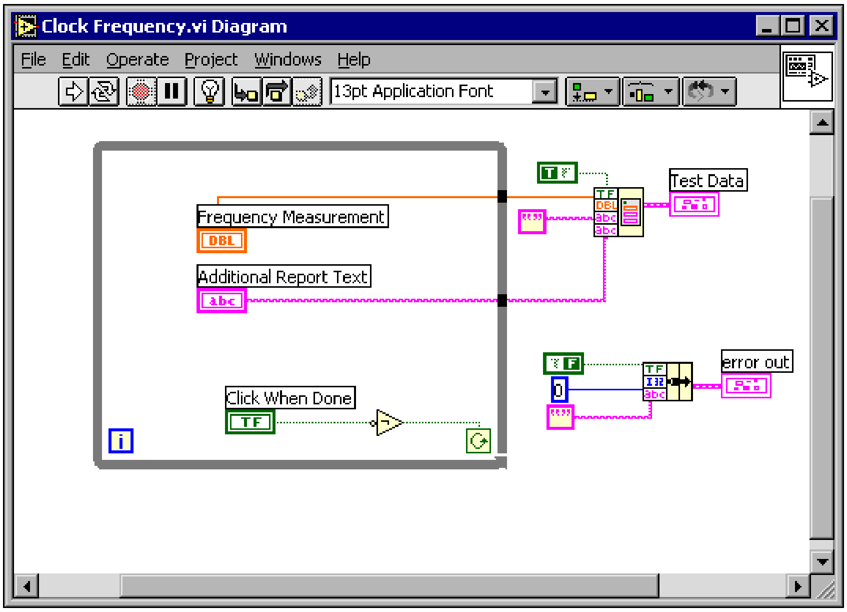
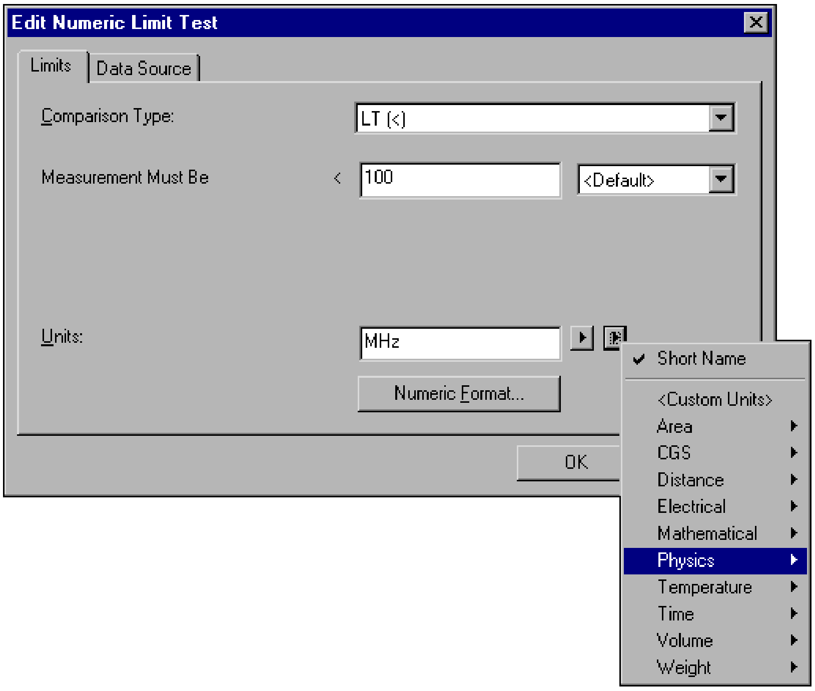


Figure 6-6. Clock Frequency.vi Block Diagram

When you run the VI, the VI loops until you enter values in the Frequency and Additional Report Text controls and click on the **Return** button.

12. After you finish building the VI, save it by selecting **File»Save** in LabVIEW.
13. Close the VI diagram and front panel.
14. Return to the sequence editor, and close the Edit LabVIEW VI Call dialog box by clicking on **OK**.
15. Right click on the Clock Frequency Test and select the **Edit Limits** command from the context menu, which displays the Edit Numeric Limit Test dialog box.
16. Set the Comparison Type control to `LT (<)` and the value to `100`, as shown in Figure 6-7.
17. Since this step simulates measuring clock frequency of a motherboard, change the units to `MHz`. Use the drop-down rings adjacent to the units control to select `Hertz` as the units and `Mega` as the units prefix. The Units control should now show the value of `megahertz`. Select the Short Name item in each ring to use the short names. The Units control

should now show the value of `MHz`. The units you specify appear in both the report and the result database. The units and units prefix are for display and documentation purposes and do not scale the measured value or affect the limit comparison.



Comparison Type: LT (<)

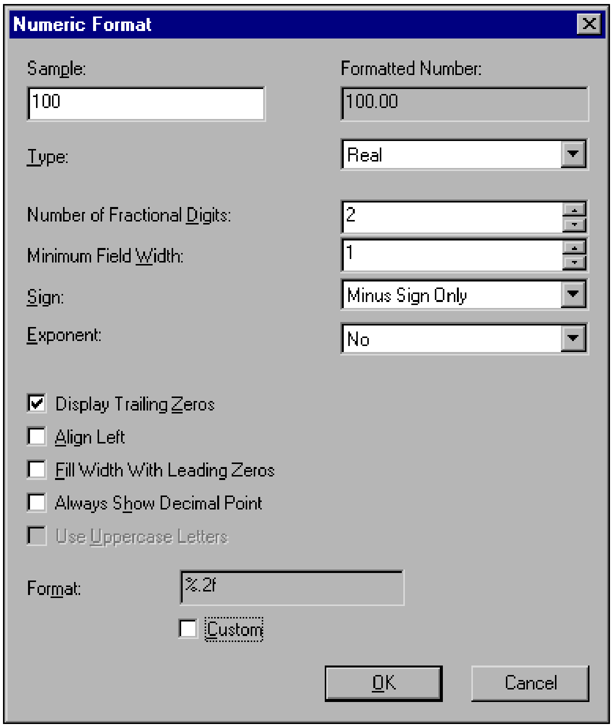
Measurement Must Be < 100 <Default>

Units: MHz

Figure 6-7. Edit Numeric Limits Test Dialog Box

18. Click on the **Numeric Format** button to open the Numeric Format dialog box. Set the control values of this dialog box to those shown in Figure 6-8. These settings specify the format of the step measurement and limit values. The format applies to the limit values that appear in the Edit Numeric Limit Test dialog box, the step description, and the test report.

    With these settings, TestStand compares the numeric measurement value that the VI returns to the constant value of 100. If the comparison is `True`, the step passes; otherwise, the step fails.


Numeric Format

Sample: 100

Formatted Number: 100.00

Type: Real

Number of Fractional Digits: 2

Minimum Field Width: 1

Sign: Minus Sign Only

Exponent: No

- [x] Display Trailing Zeros
- [ ] Align Left
- [ ] Fill Width With Leading Zeros
- [ ] Always Show Decimal Point
- [ ] Use Uppercase Letters

Format: %.2f

- [ ] Custom

Figure 6-8. Numeric Format Dialog Box

19. Click on **OK** twice to close the Numeric Format dialog box and the Edit Numeric Limit Test dialog box.
20. Save the sequence by selecting **File»Save As**. Save the sequence as `Sample5.seq` in the `TestStand\Tutorial` directory.
21. Execute the sequence by selecting **Execute»Single Pass**. When TestStand executes the Clock Frequency Test step, the VI front panel appears and runs the VI.
22. Type a numeric value of `20` in the Frequency Measurement control.
23. Type any text in the Additional Report Text control.
24. Click on the **Return** command button to return from the VI back to the sequence execution.
25. When the sequence completes the execution, examine the test report. Notice the status, measurement, and report text values for the Clock Frequency step.
26. Close the Execution window.

Debugging a Virtual Instrument Code Module

TestStand not only allows you to debug sequences, but also to step into debuggable LabVIEW VIs. In this exercise, you learn how to debug a LabVIEW test VI while executing a sequence in the sequence editor.

1. Set a breakpoint on the Clock Frequency Test step by right clicking on the step name and selecting **Toggle Breakpoint,** or by clicking to the left of the steps icon.
2. Execute the sequence by selecting **Execute»Single Pass**.
3. Click on **OK** on the Test Simulator prompt. The execution then pauses on the Clock Frequency Test step.
4. Click on the **Step Into** toolbar button, which displays the `Clock Frequency.vi` front panel in LabVIEW. The VI test is now in a paused state.
5. In LabVIEW, click on the Run toolbar button to execute the VI.
6. Select **Windows»Show Diagram** in the LabVIEW window to show the block diagram of the VI.
7. Click on the **Highlight Execution** toolbar button, shown in Figure 6-9, to highlight the flow of execution within the VI. You can set breakpoints and probes within the VI for more detailed debugging.

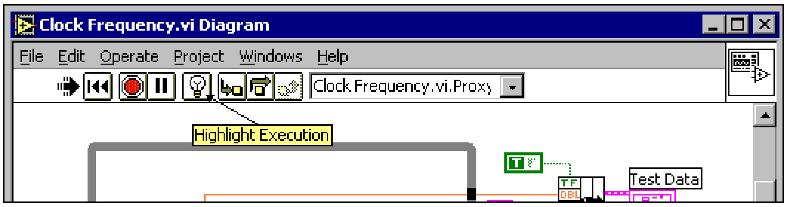

Clock Frequency.vi Diagram
File Edit Operate Project Windows Help
Clock Frequency.vi.Proxy
Highlight Execution
Test Data

Figure 6-9. LabVIEW Highlight Execution Mode

8. Click on the **Highlight Execution** toolbar button once again to turn off execution highlighting.
9. Return to the front panel by selecting **Windows»Show Panel**.
10. Type a value of `200` in the Frequency Measurement control.
11. Type any text in the Additional Report text control.
12. Click on the **Return** command button to stop the VI.

13. Click on the **Return to Caller** toolbar button to return to the sequence execution with the values in the TestData control and error out indicator.

    After the Clock Frequency Test step executes, TestStand suspends the sequence execution on the Reset Loop Index step. Notice that the status of the Clock Frequency Test step is `Failed` as expected.

Note If you do not return to caller, TestStand cannot continue the sequence execution. A common mistake is to switch from debugging a VI to the sequence editor without returning to caller. In this case, sequence execution appears to hang.

14. Select **Debug»Resume** to complete the execution.
15. Save `Sample5.seq` and close the Execution window and the Sequence File window.

When performing a task inside of a step, such as displaying a dialog box, you should monitor the state of the current TestStand execution. If the execution is terminating, you should abort the task you are performing within the VI. This functionality is implemented in the solution for this example, `<TestStand>\Tutorial\Solution\ Clock Frequency.vi`.

Debugging a LabVIEW DLL Function Using the DLL Flexible Prototype Adapter

In this exercise, you learn how to create a LabVIEW DLL test module that is called using the DLL Flexible Prototype Adapter. You also learn how to debug the module using the TestStand operator interface developed in LabVIEW. This exercise requires LabVIEW version 6*i* or later, and assumes a general familiarity with the LabVIEW development environment.

If you are not using LabVIEW, but you do use LabWindows/CVI or create C-style DLLs, you can skip this section and proceed to the *Debugging a LabWindows/CVI DLL Using the C/CVI Standard Prototype Adapter* section in this chapter.

Creating the Virtual Instrument Code

In this exercise, you complete a LabVIEW VI and use it to build a DLL code module you will call from a sequence in TestStand. This DLL function prompts the operator for a numeric value and additional report text that is passed back to TestStand.

1. Launch the LabVIEW development environment by selecting **Programs»National Instruments»LabVIEW 6»LabVIEW** from the Start menu.
2. In LabVIEW, open `<TestStand>\Tutorial\Clock Frequency Function.vi`. Switch to the diagram of this VI that has been partially completed.

   This exercise contains code that monitors the state of the current TestStand execution. When performing a task inside of a step, such as displaying a dialog box, you should monitor the execution state. If the execution is being terminated or aborted, you should abort the task you are performing within the VI. This functionality is implemented in LabVIEW using the following VIs located in the TestStand function palette: `InitializeTerminationMonitor.vi`, `GetMonitorStatus.vi`, and `CloseTerminationMonitor.vi`.
3. Switch to the front panel of the VI and complete it as shown in Figure 6-10. The Sequence Context control and indictors on the right will be used to pass data into and out of the code module, respectively. Unlike the LabVIEW Standard Prototype Adapter, you must define the parameters of the code module when using the DLL Flexible Prototype Adapter.

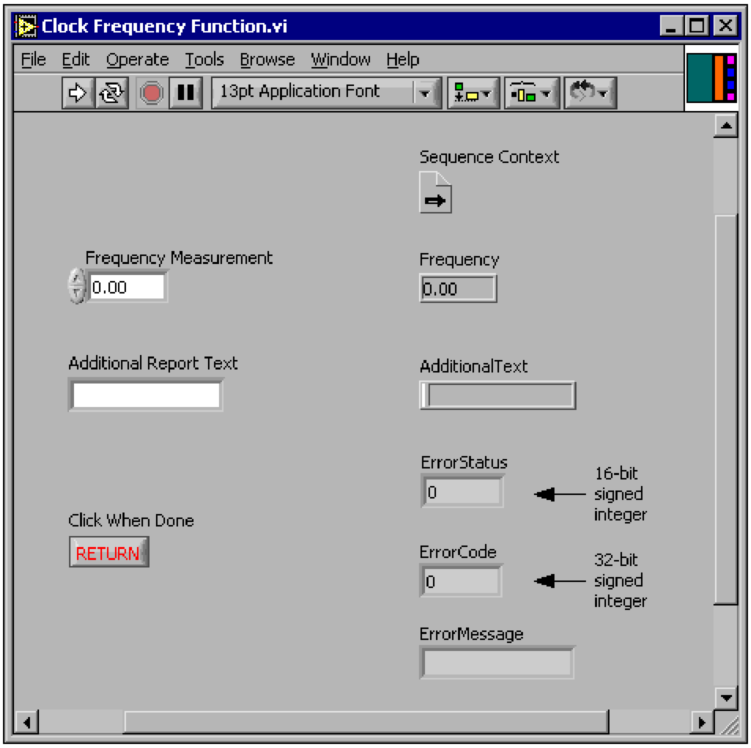


Figure 6-10. Clock Frequency Function Front Panel

4. Save the VI as `ClockFrequencyFunction.vi` by selecting **File»Save As** in LabVIEW.
5. You must wire the control and the five indicators on the right of the front panel to the VI's connector pane. Right-click on VI icon in the top right corner of the Front Panel and select **Show Connector**. Using your wiring tool, assign a terminal to the control and each indictor as shown in Figure 6-11. The arrangement and order of the connectors is not important.
6. Complete the VI diagram as shown in Figure 6-11. Notice that the While Loop halts if the operator clicks on the return button, if an error occurs within the VI, or if the current TestStand execution is terminated or aborted. The frequency measurement, additional report text, and error information are returned to TestStand as parameters of the DLL function.

Note LabVIEW does not currently create a DLL type library when one of the function parameters is a Boolean. A type library allows TestStand to obtain the parameter information for DLL functions. To simplify calling your function, convert the error status from a Boolean to a 16-bit integer, as shown in Figure 6-11. Return this integer as the error status function parameter.

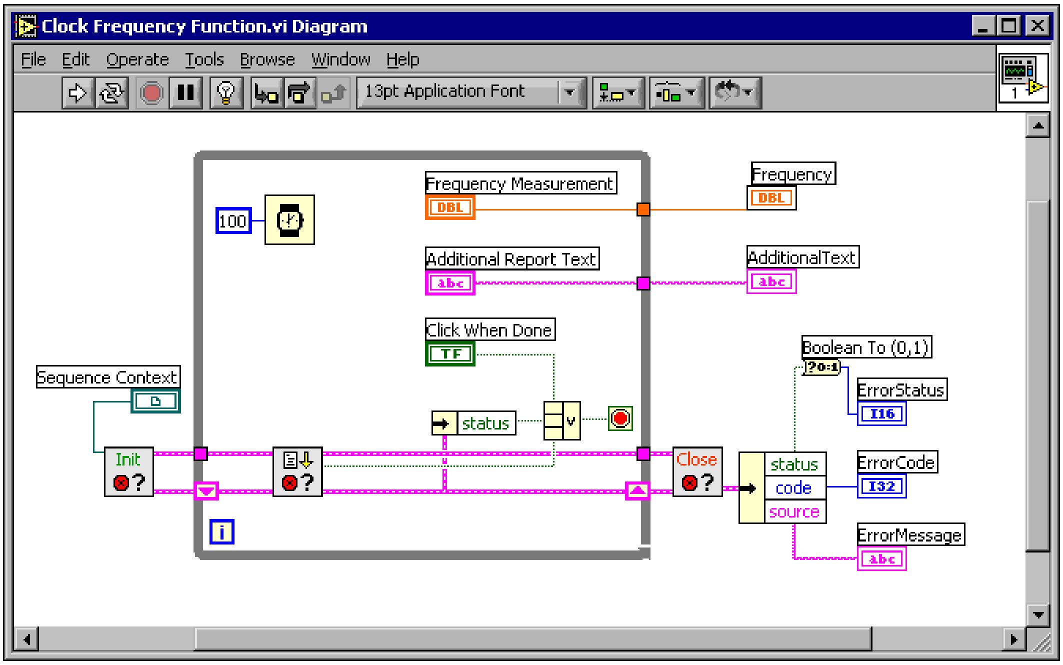
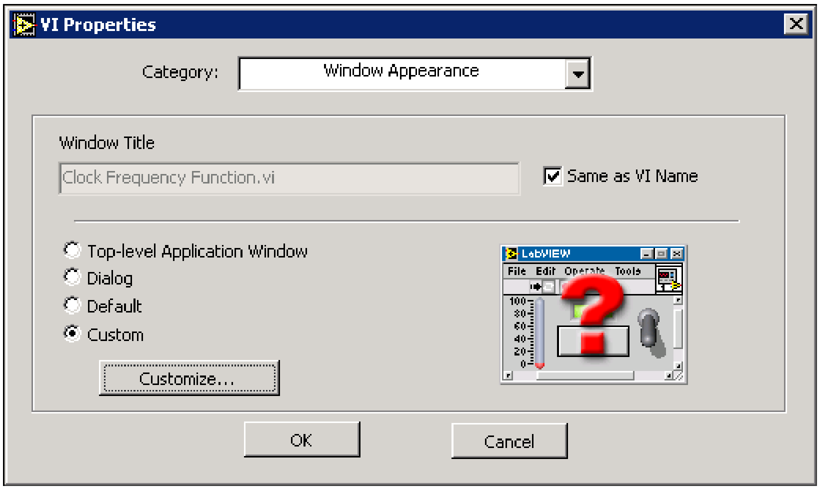


Figure 6-11. Clock Frequency Function Diagram

7. Save the VI.
8. Configure the VI settings so that the VI displays its front panel when called. Right-click on the VI icon in top right corner of the diagram window and select **VI Properties**.
9. In the VI Properties dialog box, select Window Appearance from the Category ring control, as shown in Figure 6-12. Click the **Customize** button.


VI Properties

Category: Window Appearance

Window Title

Clock Frequency Function.vi

[✓] Same as VI Name

( ) Top-level Application Window
( ) Dialog
( ) Default
(●) Custom

Customize...

OK Cancel

Figure 6-12. VI Properties Dialog Box

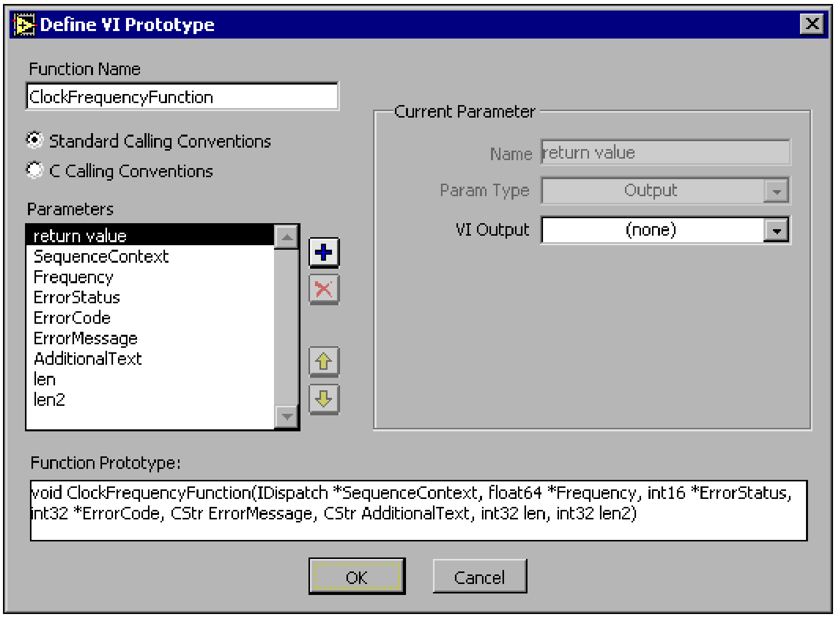
10. In the Customize Window Appearance dialog box, enable the options **Show Front Panel When Called** and **Close Afterwards if Originally Closed**. Click **OK** twice to return to the diagram window.
11. Save the VI.

Building a LabVIEW DLL Code Module

Now that you have completed the `Clock Frequency Function.vi`, you must build it into a DLL function. Before you do this, close the VI.

1. In LabVIEW, open a new VI and select **Tools»Build Application or Shared Library (DLL)**.
2. Click the **Load** button on the Build Application or Shared Library (DLL) dialog box and load the build script `<TestStand>\Tutorial\Clock Frequency.bld`. Notice that the Build Target control under the Target tab is set to Shared Library (DLL).
3. Select the Source File tab. The exported VI should be `Clock Frequency Function.vi`. Highlight this entry and click on the **Define VI Prototype** button.
4. You need to add and configure the function parameters. Click the add button marked with a "+" six times, once for each control and indicator that you wired to the VI connector pane. The Parameter control should list nine parameters, as shown in Figure 6-13. The parameters need not be in the same order. LabVIEW requires the len and len2 parameters

for the two string output parameters of this VI. These tell LabVIEW the maximum string length to be returned by the DLL function.



Function Name: ClockFrequencyFunction

Parameters:
return value
SequenceContext
Frequency
ErrorStatus
ErrorCode
ErrorMessage
AdditionalText
len
len2

Current Parameter
Name: return value
Param Type: Output
VI Output: (none)

Function Prototype:
```
void ClockFrequencyFunction(IDispatch *SequenceContext, float64 *Frequency, int16 *ErrorStatus,
int32 *ErrorCode, CStr ErrorMessage, CStr AdditionalText, int32 len, int32 len2)
```

Figure 6-13. Define VI Prototype Dialog Box

If your parameter control does not show these nine items, you did not correctly wire the control and indicators to the VI connector pane. Rewire the control and indicators for the VI connector pane before continuing.

5. Use the up and down arrows and reorder the parameters to match the order of the parameters in Figure 6-13. The parameters must be in the same order in which they are called from the sequence.
6. Enable Standard Calling Conventions radio control.
7. Click **OK** to close the Define VI Prototype dialog box.
8. Click on the **Build** button of the Build Application or Shared Library (DLL) dialog box. Once the build is complete, close all LabVIEW dialog boxes and return to the sequence editor.

Note If LabVIEW returns a file permission error (Error 8) when you build the DLL, return to the sequence editor and select **File»Unload All Modules**. When you make this selection, TestStand unloads all step code modules, which includes DLLs, VIs and any other modules the adapter loads. Return to LabVIEW and rebuild the DLL.

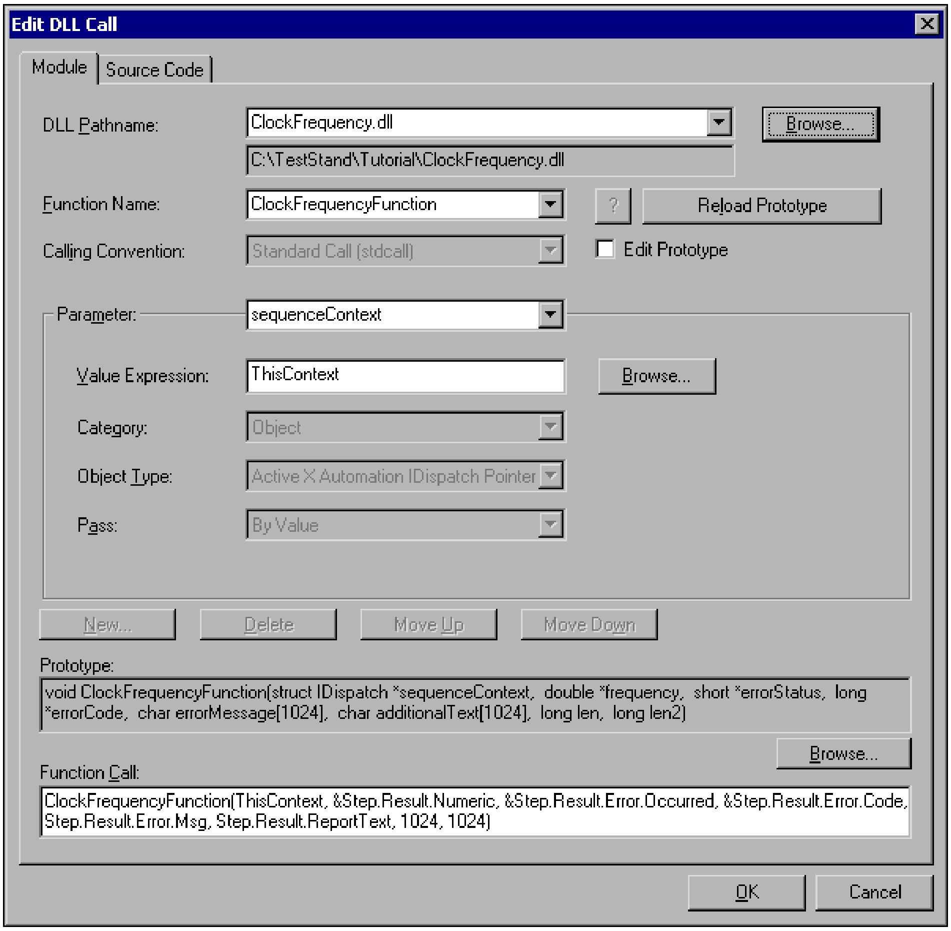
Refer to LabVIEW documentation for additional information about building DLLs in LabVIEW.

Calling the LabVIEW DLL function

Now that you have created the LabVIEW DLL function, create a step that calls this function.

1. Close all windows in the sequence editor.
2. Select **File»Open** and open the file `<TestStand>\Tutorial\Sample4.seq`, which you created in Chapter 5, *Using Variables and Properties*. You also can find this file in the `<TestStand>\Tutorial\Solution` directory.
3. Select the DLL Flexible Prototype Adapter in the Adapter Selector Ring control.
4. If you completed this or a preceding exercise, a step with the name Clock Frequency Test may already exist. Delete the step.
5. Right-click on the Power On test in the Main step group and select **Insert Step»Test»Numeric Limit Test** from the context menu.
6. Name the new step Clock Frequency Test.
7. Save the sequence as `Sample6.seq` in the `<TestStand>\Tutorial` directory by selecting **File»Save As**.
8. Right-click on the Clock Frequency Test step and select the Specify Module command in the context menu. The sequence editor displays the Edit DLL Call dialog box.
9. On the Module tab, click on the **Browse** button next to the DLL Pathname control.
10. Select the `ClockFrequency.dll` that you created.
11. Ensure that the Calling Convention control is set to Standard Call (`stdcall`).
12. Select the function `ClockFrequencyFunction` in the Function Name ring control. TestStand reads the parameter from the DLL type library and displays the function prototype, as shown in Figure 6-14.

Note TestStand may display a message saying the function does not have parameter information in the DLL. LabVIEW does not currently create a DLL type library when one of the function parameters is a Boolean. Modify your LabVIEW VI and rebuild the DLL or enter each parameter manually. If you enter the parameters manually, make sure that they match the function prototype that you created in LabVIEW when you built the DLL.



DLL Pathname: ClockFrequency.dll
C:\TestStand\Tutorial\ClockFrequency.dll
Function Name: ClockFrequencyFunction
Calling Convention: Standard Call (stdcall)
Parameter: sequenceContext
Value Expression: ThisContext
Category: Object
Object Type: Active X Automation IDispatch Pointer
Pass: By Value

Prototype:

```
void ClockFrequencyFunction(struct IDispatch *sequenceContext, double *frequency, short *errorStatus, long
*errorCode, char errorMessage[1024], char additionalText[1024], long len, long len2)
```

Function Call:

```
ClockFrequencyFunction(ThisContext, &Step.Result.Numeric, &Step.Result.Error.Occurred, &Step.Result.Error.Code,
Step.Result.Error.Msg, Step.Result.ReportText, 1024, 1024)
```

Figure 6-14. Edit DLL Call Dialog Box

13. Enter the value expressions shown below for the eight function parameters.

Parameter Name	Value Expression
sequenceContext	ThisContext
Frequency	Step.Result.Numeric
errorStatus	Step.Result.Error.Ocurred
errorCode	Step.Result.Error.Code

Parameter Name	Value Expression
errorMessage	Step.Result.Error.Msg
additionalText	Step.Result.ReportText
len	1024
len2	1024

Note If the Edit Prototype control is not checked then you cannot add new parameters or edit existing parameters. This control is automatically checked when you exit the Edit DLL call dialog box by clicking the OK button.

When you finish adding the parameters and their expressions, the value of the Function Call control should match the value shown below.

```
ClockFrequencyFunction(ThisContext,
    &Step.Result.Numeric,
    &Step.Result.Error.Occurred,
    &Step.Result.Error.Code,
    Step.Result.Error.Msg,
    Step.Result.ReportText, 1024, 1024)
```

14. Click **OK** to close the Edit DLL Call dialog box.
15. In the main sequence editor window, select **File»Save** to save the sequence file changes.
16. In the Sequence File window, right-click on the Clock Frequency Test step and select **Properties** from the context menu. On the Run Options tab, select **If Initially Active, Re-Activate When Step Completes** from the TestStand Window Activation control.
17. Click on the **OK** button to close the Clock Frequency Test Properties dialog box.
18. In the Sequence File window, right-click on the Clock Frequency Test step and select **Edit Limits** from the context menu, which displays the Edit Numeric Limit Test dialog box.
19. Set the Comparison Type control to `LT (<)` and the value to `100`, as shown in Figure 6-15.
20. Since this step simulates measuring clock frequency of a motherboard, change the units to `MHz`. Use the drop-down rings adjacent to the units control to select `Hertz` as the units and `Mega` as the units prefix. The Units control should now show the value of `megahertz`. Select the Short Name item in each ring to use the short names. The Units control should now show the value of `MHz`. The units you specify appear in

both the report and the result database. The units and units prefix are for display and documentation purposes and do not scale the measured value or affect the limit comparison.

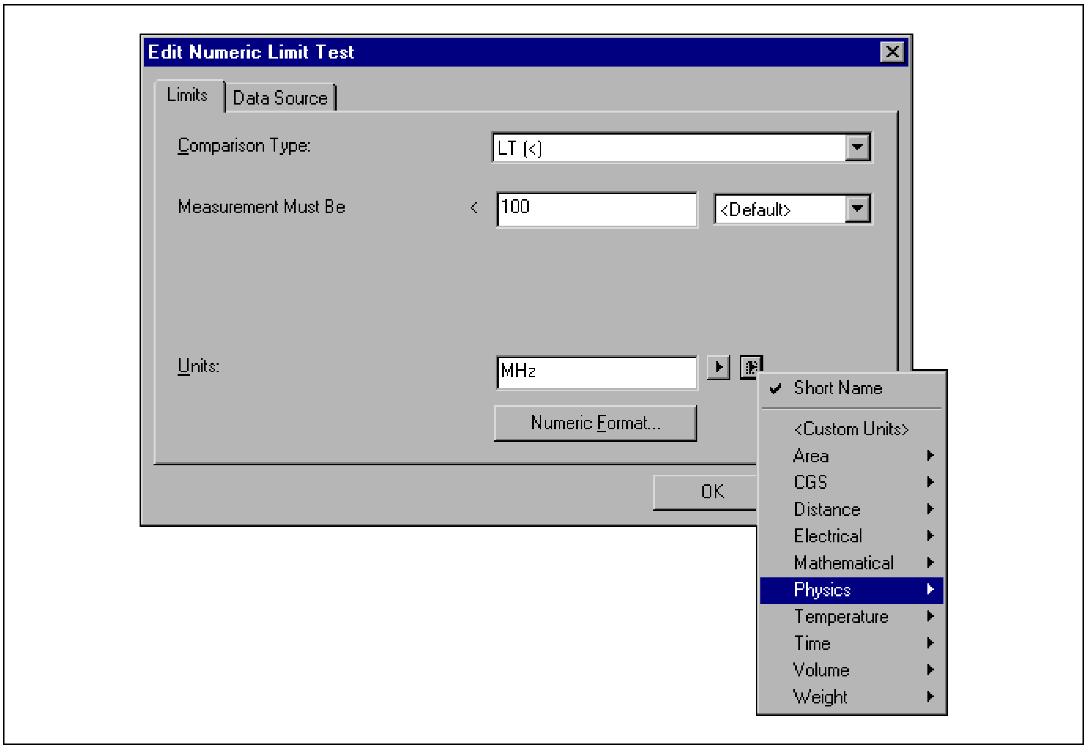
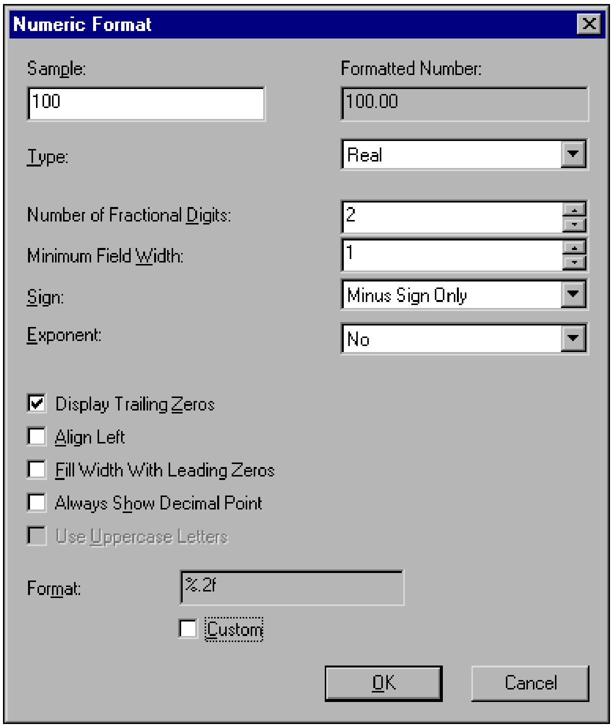


Figure 6-15. Edit Numeric Limits Test Dialog Box

21. Click on the **Numeric Format** button to open the Numeric Format dialog box. Set the control values of this dialog box to those shown in Figure 6-16. These settings specify the format of the step measurement and limit values. The format applies to the limit values that appear in the Edit Numeric Limit Test dialog box, the step description, and the test report.

    With these settings, TestStand compares the numeric measurement value that the VI returns to the constant value of 100. If the comparison is `True`, the step passes; otherwise, the step fails.



Sample: 100

Formatted Number: 100.00

Type: Real

Number of Fractional Digits: 2

Minimum Field Width: 1

Sign: Minus Sign Only

Exponent: No

[✓] Display Trailing Zeros

[ ] Align Left

[ ] Fill Width With Leading Zeros

[ ] Always Show Decimal Point

[ ] Use Uppercase Letters

Format: %.2f

[ ] Custom

Figure 6-16. Numeric Format Dialog Box

22. Click on **OK** twice to close the Numeric Format dialog box and the Edit Numeric Limit Test dialog box.
23. Save `Sample6.seq` by selecting **File»Save**.
24. Select **Execute»Single Pass** to execute the sequence.
25. Click **Done** in the Test Simulator dialog box. When TestStand executes the Clock Frequency Test step, the code module displays the user interface panel and waits for input.
26. Type a numeric value of 20 in the Frequency Measurement control.
27. Type any text in the Additional Report Text control.
28. Click the **Return** button to continue the sequence execution.
29. When the sequence completes the execution, examine the test report. Notice the values for the status, measurement, and report text for the Clock Frequency step.
30. Close the Execution window.

Debugging the DLL Function

Currently, the method for debugging a LabVIEW DLL function requires you to call the DLL from within the LabVIEW development environment. To do this, you use the LabVIEW operator interface.

1. Open the file `<TestStand>\Tutorial\Clock Frequency Function.vi` that you previously built into a DLL function.
2. Place a break point in the diagram of this VI.
3. Open and run the LabVIEW operator interface in the LabVIEW development environment by selecting **Start»Programs»National Instruments»TestStand» Operator Interfaces»LabVIEW** or by opening and running `<TestStand>\OperatorInterfaces\NI\LV TestStand - Runtime Operator Interface.vi`.

Note The VIs of the LabVIEW operator interface that ship with TestStand were written in version 5.1.1 of LabVIEW. If you have a more recent version of LabVIEW you should mass compile your VIs to decrease the loading time of these VIs.

4. After you log in, open `<TestStand>\Tutorial\Sample6.seq` within the operator interface and run it by clicking on the Single Pass button.

When the sequence step calls the DLL function, LabVIEW stops at the breakpoint you set in `Clock Frequency Function.vi`. You can use the standard LabVIEW debugging techniques to execute the VI. After you execute the VI, return to the Execution Display window of the operator interface to view the step results in the report.

Debugging a LabWindows/CVI DLL Using the C/CVI Standard Prototype Adapter

In this exercise, you learn how to create a LabWindows/CVI DLL code module that is called with the LabWindows/CVI Standard Prototype Adapter. You also learn how to debug the module by stepping into the LabWindows/CVI code from the sequence editor. This session of the tutorial assumes a general familiarity with the LabWindows/CVI development environment. If you are not using LabWindows/CVI but you create C-style DLLs, you can skip this section and proceed to *Debugging a LabWindows/CVI DLL Using the DLL Flexible Prototype Adapter*.

Note Verify that you are using the appropriate version of LabWindows/CVI with TestStand. Refer to the `readme.txt` file in the `TestStand\Doc` directory for more details.

Setting Up the Example

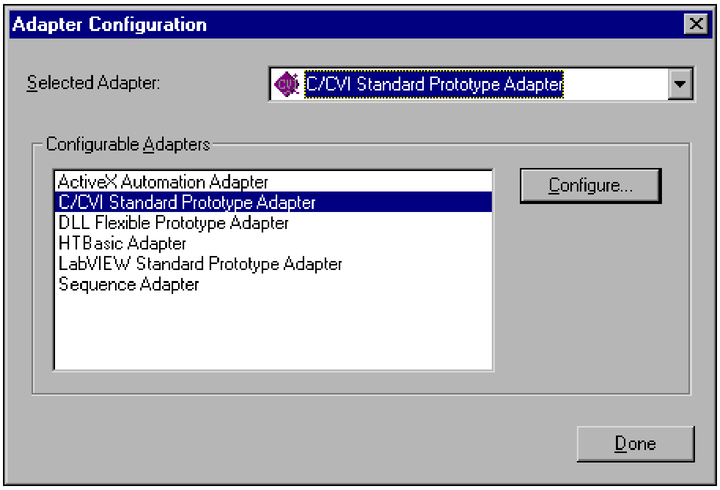
If you did not directly proceed from Chapter 5, *Using Variables and Properties*, follow these steps to set up the TestStand sequence editor so you can complete this tutorial session.

1. Close all windows in the sequence editor.
2. Select **File»Open** and open the file `<TestStand>\Tutorial\Sample4.seq`, which you created in Chapter 5, *Using Variables and Properties*. You also can find this file in the `<TestStand>\Tutorial\Solution` directory.

Creating a C/CVI Code Module Test

In this exercise, you create a LabWindows/CVI code module called using the C/CVI Standard Prototype Adapter that can prompt for a numeric value from the operator and pass the data back to TestStand.

1. Ensure the LabWindows/CVI Standard Prototype Adapter is properly configured to execute code modules in an external instance of LabWindows/CVI as follows:
   a. Select **Configure»Adapters**, which displays the Adapter Configuration dialog box, shown in Figure 6-17.


Adapter Configuration

Selected Adapter: C/CVI Standard Prototype Adapter

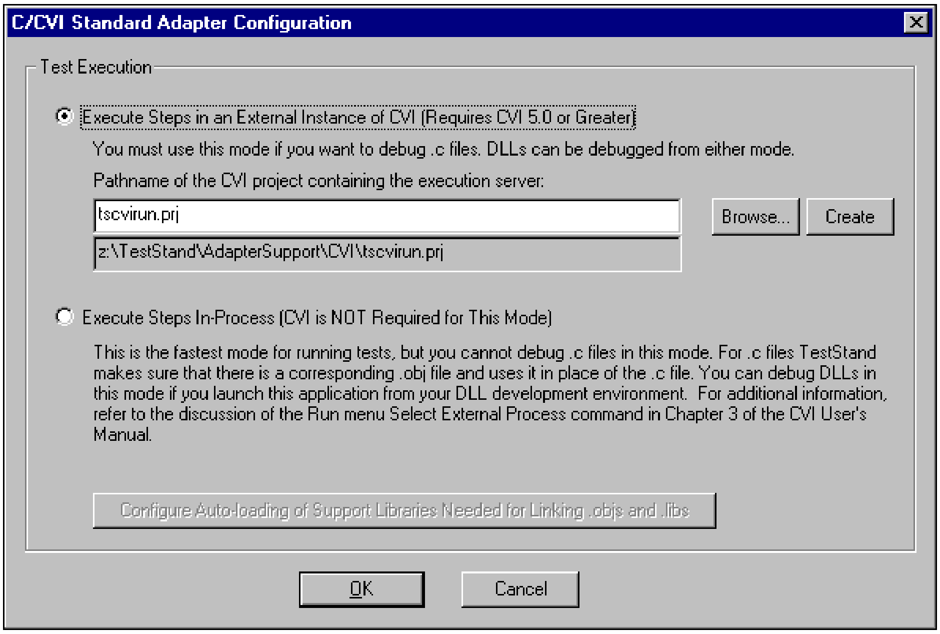
Configurable Adapters
- ActiveX Automation Adapter
- C/CVI Standard Prototype Adapter
- DLL Flexible Prototype Adapter
- HTBasic Adapter
- LabVIEW Standard Prototype Adapter
- Sequence Adapter

Figure 6-17. Adapter Configuration

b. Select the LabWindows/CVI Standard Prototype Adapter in the Configurable Adapters section.

c. Click on the **Configure** button, which displays the C/CVI Standard Adapter Configuration dialog box.

d. Verify that Execute Steps in an External Instance of CVI is enabled, and that the pathname of the LabWindows/CVI project containing the execution server is the file `tscvirun.prj` from the `TestStand\AdapterSupport\CVI` directory, as shown in Figure 6-18.


C/CVI Standard Adapter Configuration
Test Execution
Browse...
Create
Configure Auto-loading of Support Libraries Needed for Linking .objs and .libs
OK
Cancel

[✓ Execute Steps in an External Instance of CVI (Requires CVI 5.0 or Greater)]

You must use this mode if you want to debug .c files. DLLs can be debugged from either mode.

Pathname of the CVI project containing the execution server:

tscvirun.prj

z:\TestStand\AdapterSupport\CVI\tscvirun.prj

Execute Steps In-Process (CVI is NOT Required for This Mode)

This is the fastest mode for running tests, but you cannot debug .c files in this mode. For .c files TestStand makes sure that there is a corresponding .obj file and uses it in place of the .c file. You can debug DLLs in this mode if you launch this application from your DLL development environment. For additional information, refer to the discussion of the Run menu Select External Process command in Chapter 3 of the CVI User's Manual.

Figure 6-18. C/CVI Standard Adapter Configuration

e. Click on **OK** to close the C/CVI Standard Adapter Configuration dialog box. TestStand warns you that changing where tests are executed unloads all modules.

f. Click on **OK** on the Warning dialog box.

g. Click on **Done** to close the Adapter Configuration dialog box.

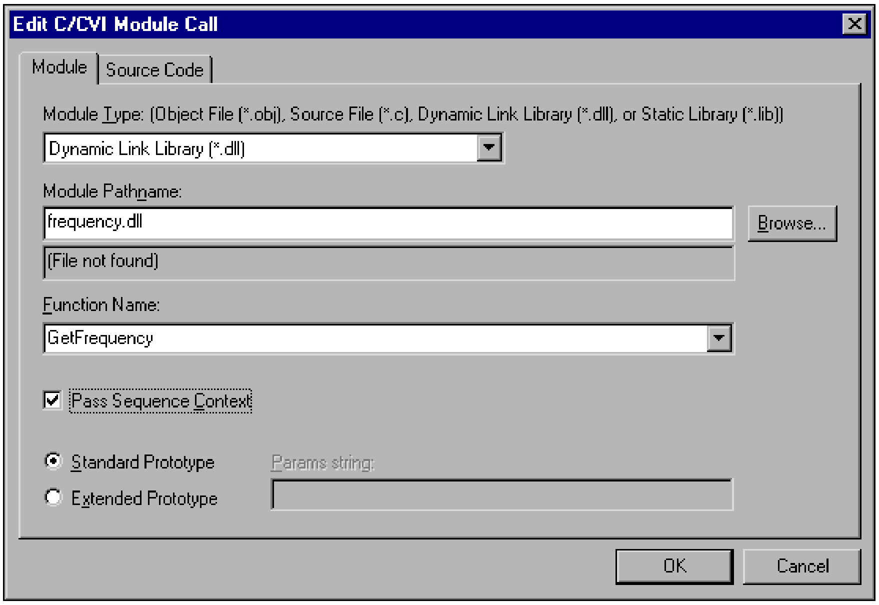
2. Verify that C/CVI Standard Prototype Adapter is selected in the Adapter Selector Ring control.

3. Right click on the Power On test in the Main step group and select **Insert Step»Tests»Numeric Limit Test** from the context menu.

4. Rename the step `Clock Frequency Test`.

5. Right click on the Clock Frequency Test and select **Specify Module** from the context menu. When you make this selection, the sequence editor displays the Edit C/CVI Module Call dialog box.
6. For the Module Type ring control, select Dynamic Link Library (`*.dll`).
7. Type the name `frequency.dll` in the Module Pathname control. The DLL file might already exist if someone previously completed this tutorial.
8. Enter the name `GetFrequency` in the Function Name control of the Edit C/CVI Module Call dialog box.
9. Enable the Pass Sequence Context checkbox.

   Figure 6-19 shows the completed Module tab.



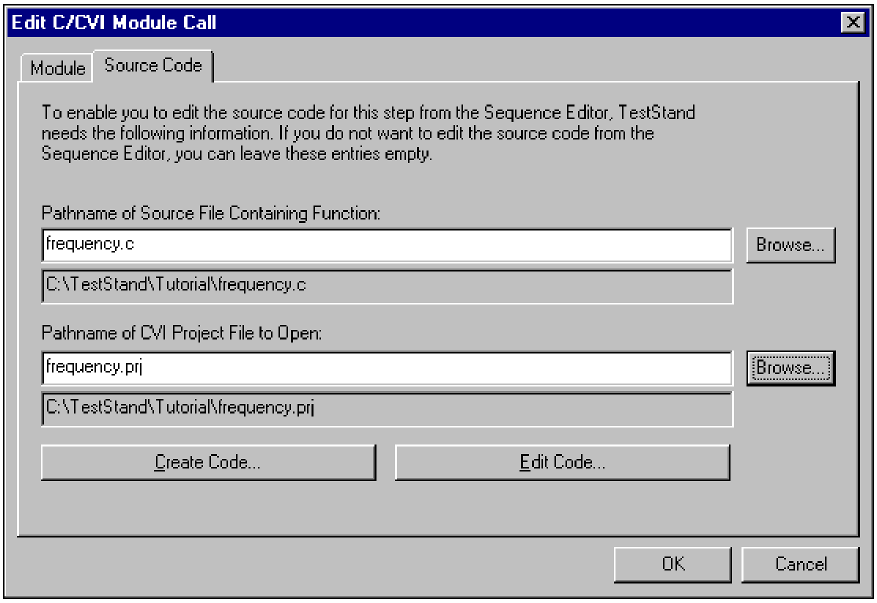
Module Type: (Object File (*.obj), Source File (*.c), Dynamic Link Library (*.dll), or Static Library (*.lib))
Dynamic Link Library (*.dll)
Module Pathname: frequency.dll
(File not found)
Function Name: GetFrequency
[✓] Pass Sequence Context
[● Standard Prototype] ○ Extended Prototype
Params string:

Figure 6-19. Edit CVI Module Call—Module Tab

10. Click on the Source Code tab. You can use the Source Code tab to generate or edit the source code for the function the step calls.
11. Click on the **Browse** button to the right of the Pathname of Source File Containing Function control, and select the `frequency.c` file in the `TestStand\Tutorial` directory.
12. Click on the **OK** button to close the Select a pathname for the source file dialog box.

13. Click on the **Browse** button to the right of the Pathname of CVI Project File to Open control, and select the `frequency.prj` file in the `TestStand\Tutorial` directory.
14. Click on the **OK** button to close the Select a pathname for the CVI project file dialog box.

    Figure 6-20 shows the completed Source Code tab for the Edit C/CVI Module Call dialog box.



To enable you to edit the source code for this step from the Sequence Editor, TestStand needs the following information. If you do not want to edit the source code from the Sequence Editor, you can leave these entries empty.

Pathname of Source File Containing Function:
frequency.c
C:\TestStand\Tutorial\frequency.c

Pathname of CVI Project File to Open:
frequency.prj
C:\TestStand\Tutorial\frequency.prj

Figure 6-20. Edit CVI Module Call—Source Code Tab

Module adapters can generate a source code shell for a step module using predefined templates. The available templates vary based on the step type and each module adapter. For each module adapter that supports source code templates, the Specify Module dialog box displays a command button for creating source code. If more than one template is associated with the step type and the specific adapter, the adapter prompts you to select a template, otherwise the adapter creates the code module from the default template.

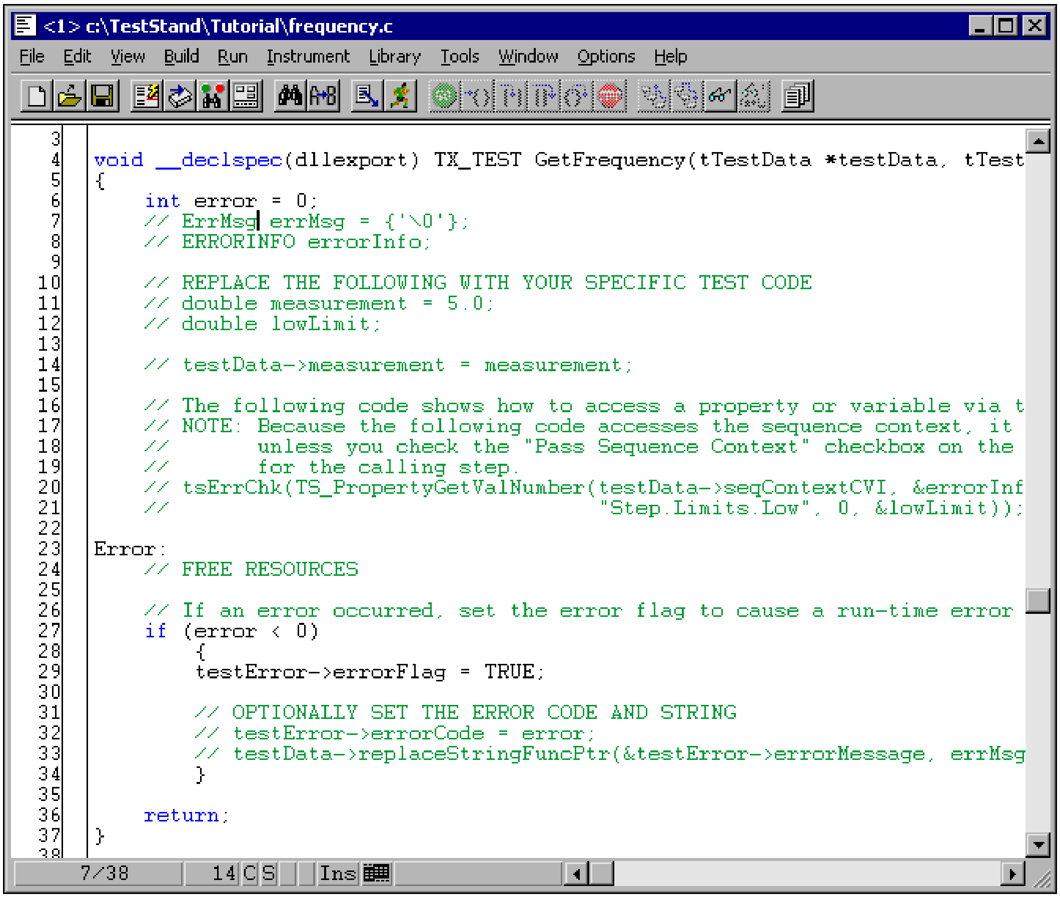
15. Click on the **Create Code** button.

    When you make this selection TestStand does the following:

    1. Launches an external instance of LabWindows/CVI
    2. Creates a new project file if one does not exist

3. Loads and saves the project file in LabWindows/CVI
4. Creates and saves a new source file if it does not exist
5. Generates and saves the source file with a template function
6. LabWindows/CVI highlights the function name

16. Figure 6-21 shows the generated function in the `frequency.c` source file.



<1> c:\TestStand\Tutorial\frequency.c
File Edit View Build Run Instrument Library Tools Window Options Help
7/38
14 C S
Ins

```
void __declspec(dllexport) TX_TEST GetFrequency(tTestData *testData, tTest
{
    int error = 0;
    // ErrMsg errMsg = {'\0'};
    // ERRORINFO errorInfo;

    // REPLACE THE FOLLOWING WITH YOUR SPECIFIC TEST CODE
    // double measurement = 5.0;
    // double lowLimit;

    // testData->measurement = measurement;

    // The following code shows how to access a property or variable via t
    // NOTE: Because the following code accesses the sequence context, it
    //       unless you check the "Pass Sequence Context" checkbox on the
    //       for the calling step.
    // tsErrChk(TS_PropertyGetValNumber(testData->seqContextCVI, &errorInf
    //                                  "Step.Limits.Low", 0, &lowLimit));

Error:
    // FREE RESOURCES

    // If an error occurred, set the error flag to cause a run-time error
    if (error < 0)
        {
        testError->errorFlag = TRUE;

        // OPTIONALLY SET THE ERROR CODE AND STRING
        // testError->errorCode = error;
        // testData->replaceStringFuncPtr(&testError->errorMessage, errMsg
        }

    return;
}
```

Figure 6-21. Generated Result from Create Code Command

Note If someone else previously completed this session of the tutorial, the function `GetFrequency` might already exist in the source file. When LabWindows/CVI prompts you to replace the existing function, click on the **Replace** button to continue with this tutorial session.

The `GetFrequency` prototype function contains two parameters, `tTestData` and `tTestError`. The LabWindows/CVI Standard Prototype Adapter uses these parameters to pass common data between TestStand and the code module. Following is a list of the different elements within these two structures and how the adapter uses them.

tTestData

- `result`—Set by test function to indicate whether the test passed.
- `measurement`—Numeric measurement that the test function returns.
- `inBuffer`—For passing a string parameter to a test function.
- `outBuffer`—Output message to display in the report.
- `modPath`—Directory path of module containing the test function.
- `modFile`—Filename of module containing the test function.
- `hook`—Reserved (no longer used).
- `hookSize`—Reserved (no longer used).
- `mallocFuncPtr`—Contains a function pointer to `malloc`, which a code module must use to allocate memory for any buffer that it assigns to the `inBuffer`, `outBuffer`, and `errorMessage` fields.
- `freeFuncPtr`—Contains a function pointer to `free`, which a code module must use to free any buffers that the `inBuffer`, `outBuffer`, and `errorMessage` fields point to.
- `seqContextDisp`—A dispatch pointer to the sequence context.
- `seqContextCVI`—A CVI ActiveX Automation handle for the sequence context.
- `stringMeasurement`—String value that the test function returns.
- `structVersion`—Structure version number.
- `replaceStringFuncPtr`—Contains a function pointer to a `ReplaceString` function, which a code module can use to change the value of any buffers that the `inBuffer`, `outBuffer`, and `errorMessage` fields point to.

tErrorData

- `errorFlag`—The test function must set this to `True` if an error occurs.
- `errorLocation`—Reserved (no longer used).

- `errorCode`—The test function can set this to a non-zero value if an error occurs.
- `errorMessage`—The test function can set this to a descriptive string if an error occurs.

Refer to Chapter 12, *Module Adapters*, in the *TestStand User Manual* for more details on these structures.

17. Update the `Frequency.c` code to prompt an operator to enter values into a frequency numeric control and a report text string control using the `frequency.uir` user interface resource file, as follows:
    a. Leaving the `Frequency.c` window open, return to the project window by selecting **Windows»Project**.
    b. Open the `Frequency.uir` file from the Project window.
    c. Right-click on the **Return** button and select **Generate Control Callback** from the context menu. If you have more than one `.c` file open, LabWindows/CVI prompts you to select a target file into which to insert the callback. Select `frequency.c` as the target file.
    d. You have just created the `ReturnCallback` function in `Frequency.c`. The `ReturnCallback` function quits the user interface. Close the UIR file after you are done, you will not be making any changes to it.
    e. Update the source code for the `GetFrequency` and `ReturnCallback` function as shown. The other functions in the source file do not need modification. Changed lines appear in bold.

```
void __declspec(dllexport) TX_TEST
    GetFrequency(tTestData   *testData, tTestError
    *testError)
{
    int error = 0;
    int panelHandle, panel, control;
    char stringBuffer[512];

    panelHandle = LoadPanelEx (0, "frequency.uir",
      PANEL, __CVIUserHInst);

    if (panelHandle < 0)
        {
        error = panelHandle;
        goto Error;
        }

    DisplayPanel (panelHandle);
```

```
        //Automatically closes user display dialog when
        //TestStand execution terminates or aborts.
        TS CancelDialogIfExecutionStops (panelHandle,
          testData->seqContextCVI);
        RunUserInterface();

        // Assign values from UIR to return data structure
        GetCtrlVal (panelHandle, PANEL_FREQUENCY,
          &testData->measurement);
        GetCtrlVal (panelHandle,
          PANEL_ADDITIONAL_REPORT, stringBuffer);
        testData->replaceStringFuncPtr(&testData->
          outBuffer, stringBuffer);

    Error:
        // FREE RESOURCES
        DiscardPanel(panelHandle);

        // If an error occurred, set the error flag to
        // cause a run-time error in TestStand.
        if (error < 0)
        {
            testError->errorFlag = TRUE;

            // OPTIONALLY SET THE ERROR CODE AND STRING
            testError->errorCode = error;
            testData->replaceStringFuncPtr(
              &testError->errorMessage, "A run-time error
              occurred.");
        }

        return;
    }
    intCVICALLBACK ReturnCallback (int panelHandle, int
        control, int event, void*callbackData, int
        eventData1, int eventData2
    {
        switch(event)
            {
            case EVENT_COMMIT:
            QuitUserInterface(0);
            break;
        }
    return 0;
}
```

18. Compile the `Frequency.c` source code by selecting **Build»Compile File** to verify that your changes are correct.

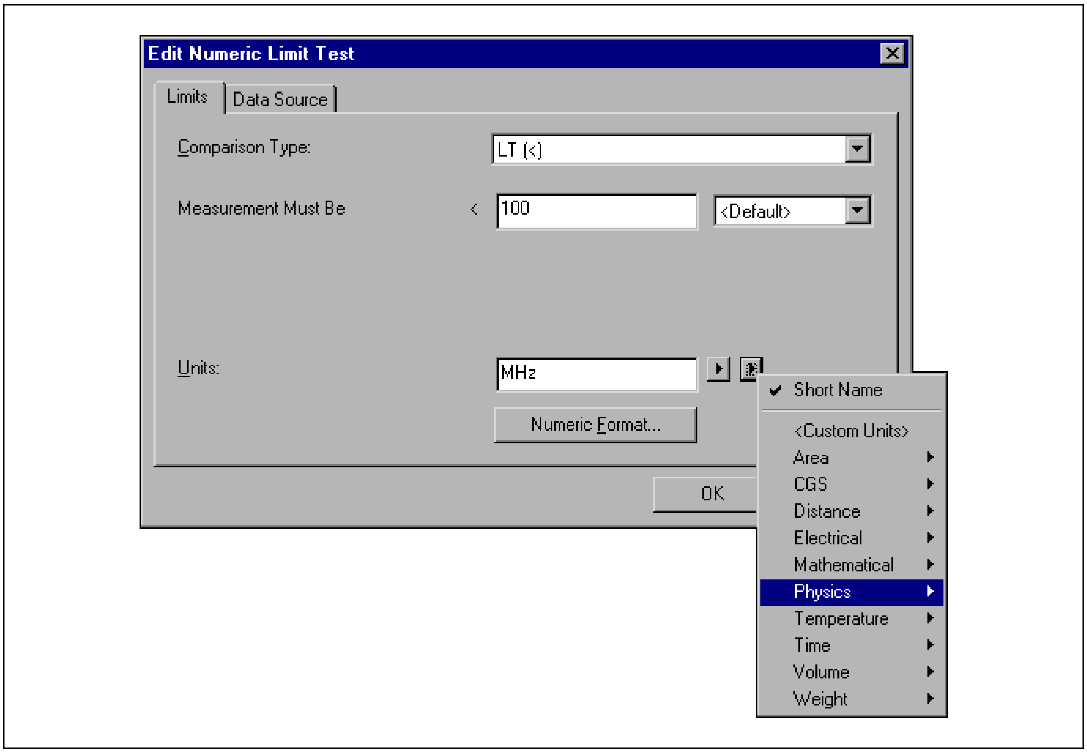
Note When performing a task within a step module, such as displaying a dialog box, monitor the status of the TestStand execution under which the step module was called. If the execution terminates or aborts, you should abort the task you are performing within the step module. The function `TS_CancelDialogIfExecutionStops()` is used in step modules that display dialog boxes that you want to close automatically when execution terminates or aborts. This function assumes that your program controls the dialog box through a call to `RunUserInterface`.

19. Save the source code after you successfully compile.
20. Rebuild the DLL by selecting **Build»Create Dynamic Link Library** in the Project window. If the DLL already exists, overwrite the existing copy.

Note If LabWindows/CVI returns a file permission error when creating the DLL, return to the sequence editor and select the Unload All Modules command from the File menu. When you make this selection, TestStand unloads all step code modules, which include DLLs, VIs and any other modules the adapter loads. Return to LabWindows/CVI and rebuild the DLL.

21. Close the external instance of LabWindows/CVI that contains the `frequency.prj` DLL project.
22. In the TestStand sequence editor, close the Edit C/CVI Module Call dialog box by clicking on the **OK** button.
23. In the Sequence File window, right click on the Clock Frequency Test and select **Edit Limits** from the context menu.

24. In the Edit Numeric Limit Test dialog box, set the comparison type to less than, `LT (<)`, and the measurement value to `100`, as shown in Figure 6-22.



Comparison Type: LT (<)

Measurement Must Be < 100 <Default>

Units: MHz

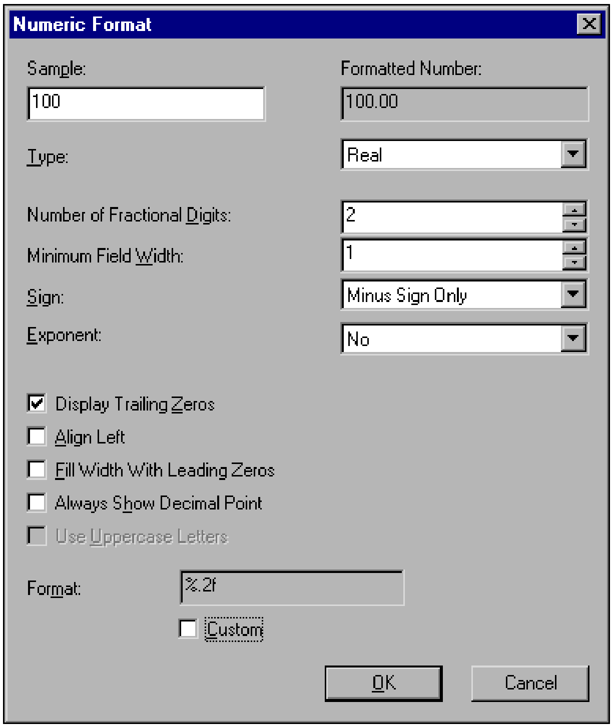
Figure 6-22. Edit Numeric Limits Test Dialog Box

25. Since this step simulates measuring clock frequency of a motherboard, change the units to `MHz`. Use the drop-down rings adjacent to the units control to select `Hertz` as the units and `Mega` as the units prefix. The Units control should now show the value of `megahertz`. Select the Short Name item in each ring to use the short names. The Units control should now show the value of `MHz`. The units you specify appear in both the report and the result database. The units and units prefix are for display and documentation purposes and do not scale the measured value or affect the limit comparison.

26. Click on the **Numeric Format** button to open the Numeric Format dialog box. Set the control values of this dialog box to those shown in Figure 6-23. These settings specify the format of the step measurement and limit values. The format applies to the limit values that appear in

the Edit Numeric Limit Test dialog box, the step description, and the test report.

With these settings, TestStand compares the numeric measurement value that the VI returns to the constant value of 100. If the comparison is `True`, the step passes; otherwise, the step fails.



Numeric Format

Sample: 100

Formatted Number: 100.00

Type: Real

Number of Fractional Digits: 2

Minimum Field Width: 1

Sign: Minus Sign Only

Exponent: No

- [x] Display Trailing Zeros
- [ ] Align Left
- [ ] Fill Width With Leading Zeros
- [ ] Always Show Decimal Point
- [ ] Use Uppercase Letters

Format: %.2f

- [ ] Custom

Figure 6-23. Numeric Format Dialog Box

27. Click on **OK** twice to close the Numeric Format dialog box and the Edit Numeric Limit Test dialog box.
28. Save the sequence by selecting **File»Save As**. Save the sequence as `Sample7.seq` in the `TestStand\Tutorial` directory.
29. Execute the sequence by selecting **Execute»Single Pass**. TestStand launches a new external instance of LabWindows/CVI to execute steps.
30. Click on **Done** in the Test Simulator dialog box.

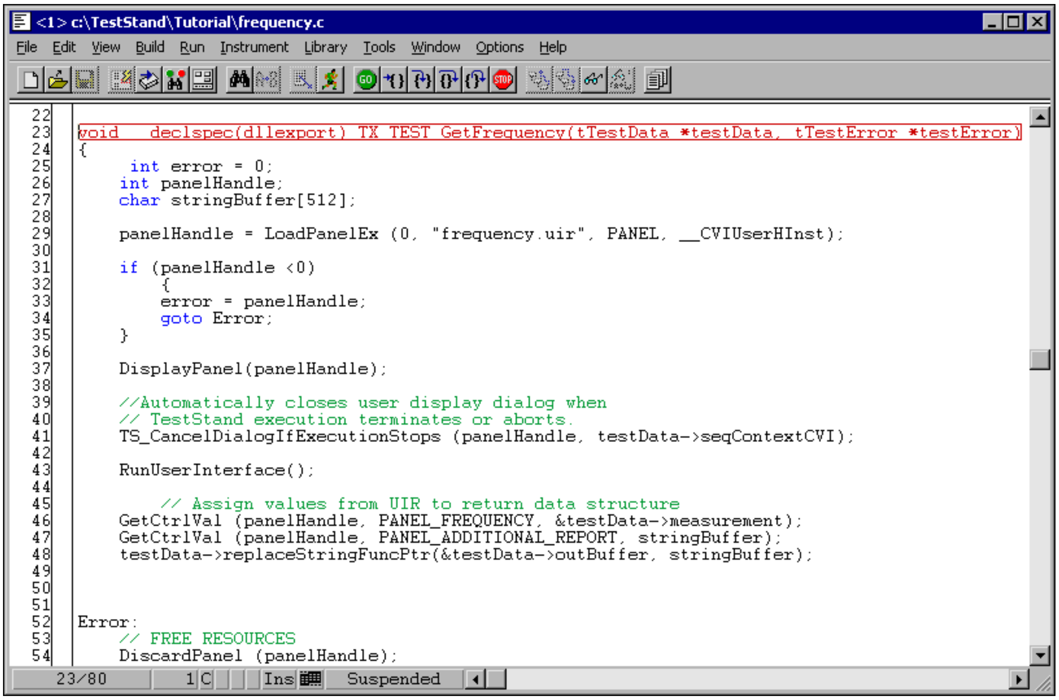
    When TestStand executes the Clock Frequency Test step, the code module displays the user interface panel and waits for input.

31. Type a numeric value of `20` in the Frequency Measurement control.
32. Type any text in the Additional Report Text control.
33. Click on the **Return** button to continue the sequence execution.
34. When the sequence completes the execution, examine the test report. Notice the values for the status, measurement, and report text for the Clock Frequency step.
35. Close the Execution window.

Debugging a CVI Code Module

TestStand not only allows you to debug sequences, but also to step directly into debuggable LabWindows/CVI code modules. In this exercise, you examine how to debug a LabWindows/CVI code module while executing a sequence in the sequence editor.

1. Set a breakpoint on the Clock Frequency Test step by right clicking on the step name and selecting **Toggle Breakpoint**. Remember that a breakpoint is enabled for a step if a stop sign icon is visible to the left of the step name in the sequence window.
2. Execute the sequence by selecting **Execute»Single Pass**.
3. Click on **Done** on the Test Simulator prompt. The execution pauses on the Clock Frequency Test step.
4. Click on the **Step Into** toolbar button, which activates an external instance of LabWindows/CVI and enters a breakpoint state on the `GetFrequency` function, as shown in Figure 6-24.


<1> c:\TestStand\Tutorial\frequency.c
File Edit View Build Run Instrument Library Tools Window Options Help

```
void __declspec(dllexport) TX_TEST GetFrequency(tTestData *testData, tTestError *testError)
{
     int error = 0;
    int panelHandle;
    char stringBuffer[512];

    panelHandle = LoadPanelEx (0, "frequency.uir", PANEL, __CVIUserHInst);

    if (panelHandle <0)
        {
        error = panelHandle;
        goto Error;
    }

    DisplayPanel(panelHandle);

    //Automatically closes user display dialog when
    // TestStand execution terminates or aborts.
    TS_CancelDialogIfExecutionStops (panelHandle, testData->seqContextCVI);

    RunUserInterface();

        // Assign values from UIR to return data structure
    GetCtrlVal (panelHandle, PANEL_FREQUENCY, &testData->measurement);
    GetCtrlVal (panelHandle, PANEL_ADDITIONAL_REPORT, stringBuffer);
    testData->replaceStringFuncPtr(&testData->outBuffer, stringBuffer);



Error:
    // FREE RESOURCES
    DiscardPanel (panelHandle);
```

Figure 6-24. Stepping into the GetFrequency Function

5. Using the **Run»Step Over** command in the menu in LabWindows/CVI, step through the code module.

Note The LabWindows/CVI debugging window remains in the foreground while you are debugging. Switch to the Clock Frequency dialog box to enter values into the controls.

6. When the user interface resource displays, type a value of `200` in the Frequency Measurement control.
7. Type any text into the Additional Report Text control.
8. Click on the **Return** button.
9. Exit the function by selecting **Run»Finish Function** to return to the sequence execution.
10. After the Clock Frequency Test step executes, TestStand suspends the sequence execution on the Reset Loop Index step. Notice that the status of the Clock Frequency Test step is `Failed` as expected.
11. Select **Debug»Resume** to complete the sequence execution.

12. Close the Execution window and the Sequence File window.
13. If TestStand prompts you to save the sequence file, save the sequence as `Sample7.seq` in the `TestStand\Tutorial` directory.

Debugging a LabWindows/CVI DLL Using the DLL Flexible Prototype Adapter

In this exercise, you create a LabWindows/CVI DLL code module and call it using the DLL Flexible Prototype Adapter. You also learn how to debug this module by launching the sequence editor from the LabWindows/CVI development environment.

While this exercise uses LabWindows/CVI to create and debug the code module, the information about the DLL Flexible Prototype Adapter and the debugging techniques are applicable when you call C-style DLLs written in other application development environments. If you are not using LabWindows/CVI and do not create C-style DLLs, you can skip this section and proceed to Chapter 7, *Using Run-Time Operator Interfaces*.

Setting Up the Example

If you did not directly proceed from Chapter 5, *Using Variables and Properties*, follow these steps to set up the TestStand sequence editor so you can complete this tutorial session.

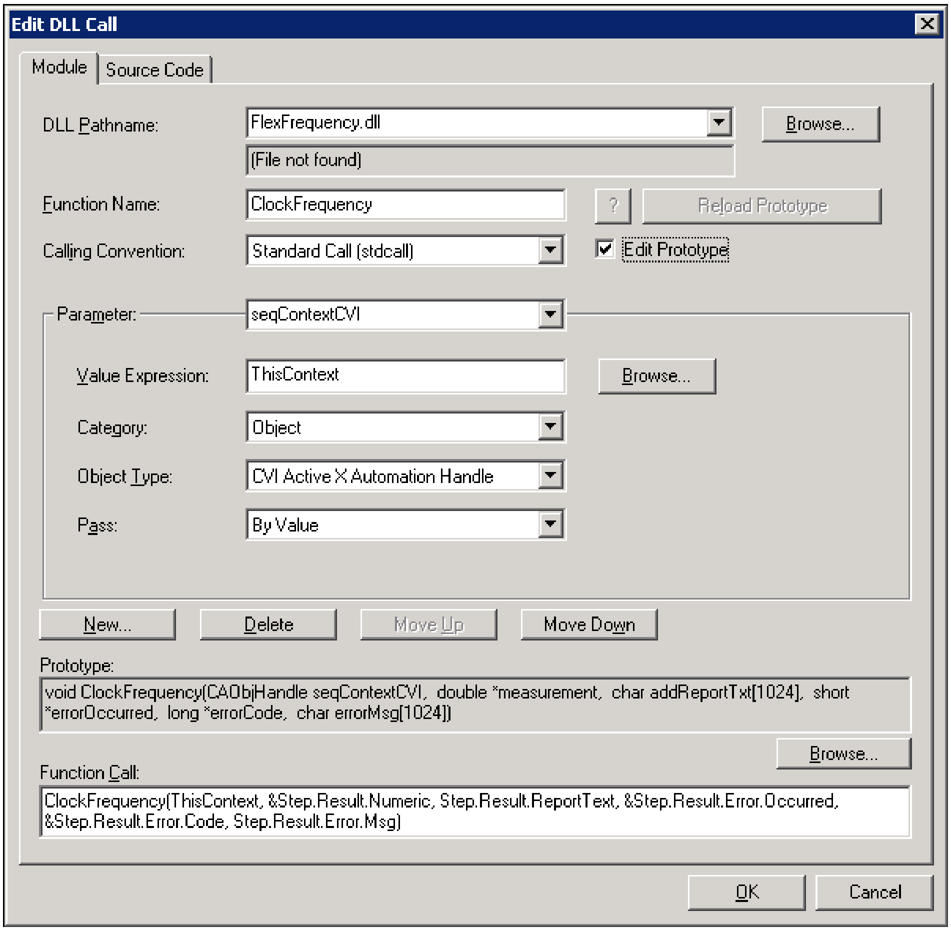
1. Close all windows in the sequence editor.
2. Select **File»Open** and open the file `<TestStand>\Tutorial\Sample4.seq`, which you created in Chapter 5, *Creating and Debugging Tests*. You also can find this file in the `<TestStand>\Tutorial\Solution` directory.

Creating the LabWindows/CVI Code Module

In this exercise, you create a LabWindows/CVI code module that can prompt the operator for a numeric value and pass the data back to TestStand.

1. Make sure that the DLL Flexible Prototype Adapter is selected in the Adapter Selector Ring control.
2. Right-click on the Power On test in the Main step group and select **Insert Step»Tests»Numeric Limit Test** from the context menu.
3. Rename the step Clock Frequency Test.

4. Right-click on the Clock Frequency Test step and select **Specify Module** from the context menu, which displays the Edit DLL Call dialog box as shown in Figure 6-25.



DLL Pathname: FlexFrequency.dll

(File not found)

Function Name: ClockFrequency

Calling Convention: Standard Call (stdcall)

☑ Edit Prototype

Parameter: seqContextCVI

Value Expression: ThisContext

Category: Object

Object Type: CVI ActiveX Automation Handle

Pass: By Value

Prototype:

```
void ClockFrequency(CAObjHandle seqContextCVI, double *measurement, char addReportTxt[1024], short
*errorOccurred, long *errorCode, char errorMsg[1024])
```

Function Call:

```
ClockFrequency(ThisContext, &Step.Result.Numeric, Step.Result.ReportText, &Step.Result.Error.Occured,
&Step.Result.Error.Code, Step.Result.Error.Msg)
```

Figure 6-25. Edit DLL Dialog Box for a LabWindows/CVI Code Module

5. On the Module tab, type `FlexFrequency.dll` in the DLL Pathname control.
6. Type `ClockFrequency` in the Function Name control.
7. Ensure that the Calling Convention control is set to Standard Call (`stdcall`).
8. Unlike the C/CVI Standard Prototype Adapter, you must define the parameters of the code module when using the DLL Flexible Prototype

Adapter. You must enter the desired parameters and then have TestStand generate your C code. Click on the **New** button to begin adding the first parameter for the function.

9. Rename the arg1 parameter `seqContextCVI`. The name of the argument is not critical; you can choose the name of the argument. Avoid using spaces within the parameter names since these will be used as the parameter names within your C code.
10. Select **Object** in the Category ring control.
11. Select **CVI ActiveX Automation Handle** in the Object Type ring control.
12. Enter the expression, `ThisContext`, in the Value Expression control.
13. Set the Pass control to By Value.
14. Add the five other parameters of your DLL function in a similar manner. To add each parameter, start by clicking on the **New** button. Use the values in Table 6-1 to complete the control settings for each parameter.

Note If the Edit Prototype control is not enabled, you cannot add new parameters or edit existing parameters. This control is automatically enabled when you exit the Edit DLL call dialog box by clicking the **OK** button.

Table 6-1. Parameter Control Table of Values

Control	Value
Parameter Name:	measurement
Value Expression:	Step.Result.Numeric
Result Action:	No Action
Category:	Numeric
Data Type:	64-bit Real Number (double)
Pass:	By Reference (by pointer)
Parameter Name:	addReportTxt
Value Expression:	Step.Result.ReportText
Category:	String
Pass:	C String Buffer
Number of Elements:	1024

Table 6-1. Parameter Control Table of Values (Continued)

Control	Value
Parameter Name: Value Expression: Result Action: Category: Data Type: Pass:	errorOccurred Step.Result.Error.Occurred No Action Numeric Signed 16-bit Integer By Reference (by pointer)
Parameter Name: Value Expression: Result Action: Category: Data Type: Pass:	errorCode Step.Result.Error.Code No Action Numeric Signed 32-bit Integer By Reference (by pointer)
Parameter Name: Value Expression: Category: Pass: Number of Elements:	errorMsg Step.Result.Error.Msg String C String Buffer 1024

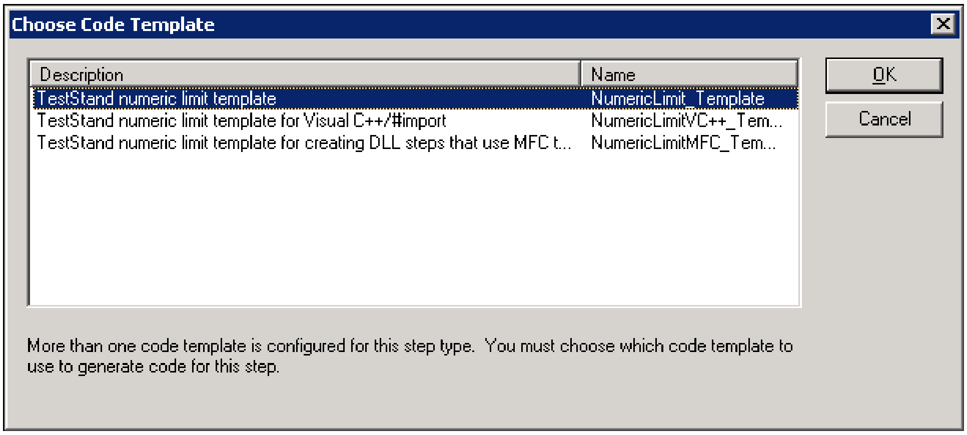
When you finish adding the parameters and their settings, the value of the Prototype control, which is automatically generated, should match the value shown below.

```
void ClockFrequency(CAObjHandle seqContextCVI, double
    *measurement, char addReportTxt[1024], short
    *errorOccurred, long *errorCode, char
    errorMsg[1024])
```

In addition, the value of the Function Call control should match the value shown below.

```
ClockFrequency(ThisContext, &Step.Result.Numeric,
    Step.Result.ReportText,
    &Step.Result.Error.Occurred,
    &Step.Result.Error.Code,
    Step.Result.Error.Msg)
```

15. Select the Source Code tab of the Edit DLL Call dialog box and type `FlexFrequency.c` as the value of the Pathname of Source File Containing Function control. Directly beneath this control you should see the message (File not Found). If the file is found, then either another user has completed the exercise, or the `<TestStand>/Tutorial/Solutions` directory has been added as one of the TestStand search directories. In this case, choose a different file name.
16. Click the **Create Code** button.
17. Choose a pathname for the source file. Browse to the `<TestStand>/Tutorial` directory and click the **OK** button, which displays the Choose Code Template dialog box shown in Figure 6-26.



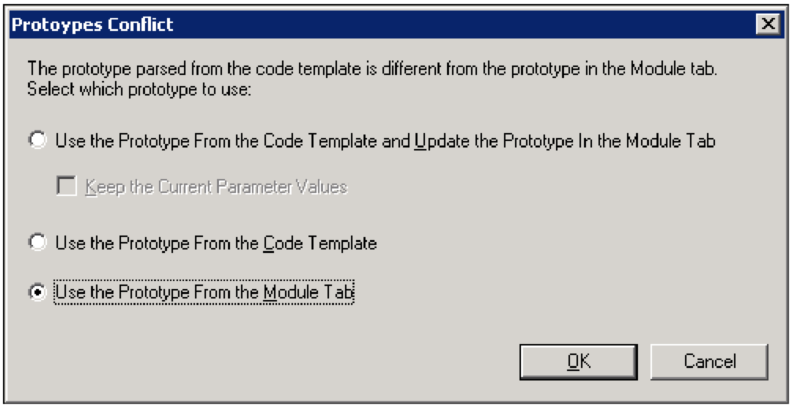
Description	Name
TestStand numeric limit template	NumericLimit_Template
TestStand numeric limit template for Visual C++/#import	NumericLimitVC++_Tem...
TestStand numeric limit template for creating DLL steps that use MFC t...	NumericLimitMFC_Tem...

More than one code template is configured for this step type. You must choose which code template to use to generate code for this step.

Figure 6-26. Choose Code Template Dialog Box

18. Select the first option, TestStand numeric limit template – NumericLimit_Template, and click the **OK** button.
19. The template is based on an existing `.c` file that ships with TestStand. The function within this `.c` file has default parameters that are different than those you have just specified. TestStand prompts you to

resolve the differences in the parameters by displaying a Prototypes Conflict dialog box shown in Figure 6-27.



The prototype parsed from the code template is different from the prototype in the Module tab. Select which prototype to use:

( ) Use the Prototype From the Code Template and Update the Prototype In the Module Tab

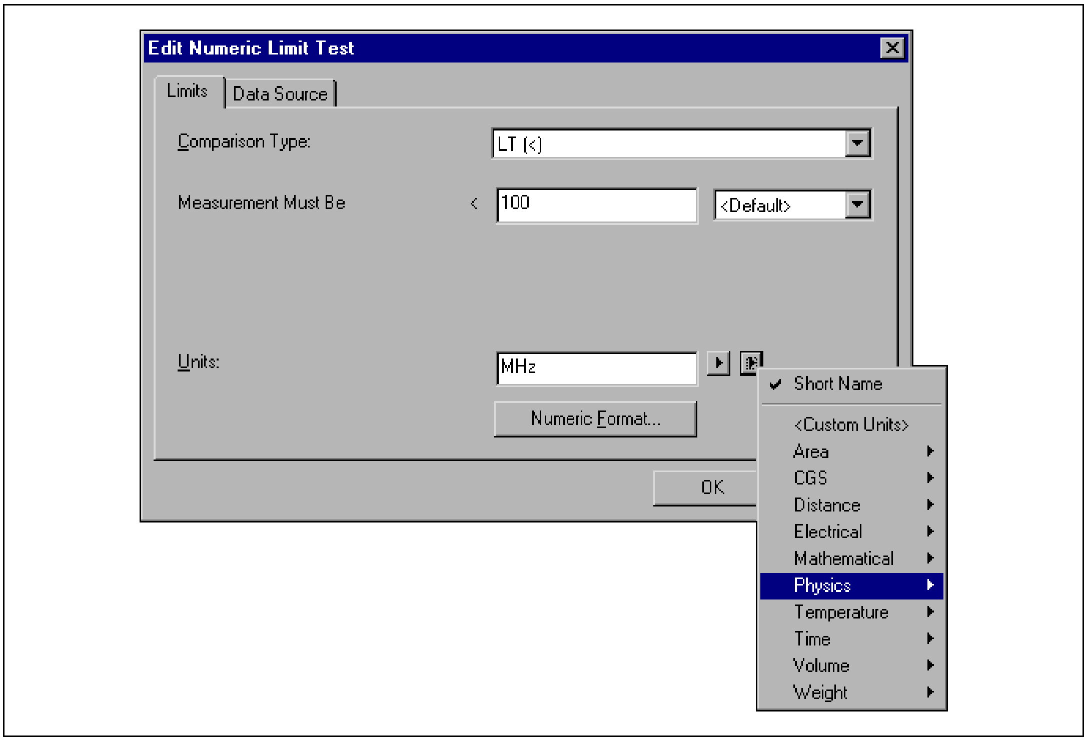
[ ] Keep the Current Parameter Values

( ) Use the Prototype From the Code Template

(●) Use the Prototype From the Module Tab

Figure 6-27. Prototypes Conflict Dialog Box

20. Select **Use Prototype From the Module Tab** and then click the **OK** button.
21. TestStand either opens the `.c` file with an application on your system that is registered to open files with a `*.c` extension, or it prompts you to launch Notepad to view the newly created `.c` file. After viewing the `.c` file, return to the sequence editor and click the **OK** button to close the Edit DLL Call dialog box.
22. In the Sequence File window, right-click on the Clock Frequency Test and select Edit Limits from the context menu, which displays the Edit Numeric Limit Test dialog box.
23. Set the Comparison Type control to `LT (<)` and the value to `100`, as shown in Figure 6-28.
24. Since this step simulates measuring clock frequency of a motherboard, change the units to `MHz`. Use the drop-down rings adjacent to the units control to select `Hertz` as the units and `Mega` as the units prefix. The Units control should now show the value of `megahertz`. Select the Short Name item in each ring to use the short names. The Units control should now show the value of `MHz`. The units you specify appear in both the report and the result database. The units and units prefix are for display and documentation purposes and do not scale the measured value or affect the limit comparison.


Edit Numeric Limit Test

Limits | Data Source

Comparison Type: LT (<)

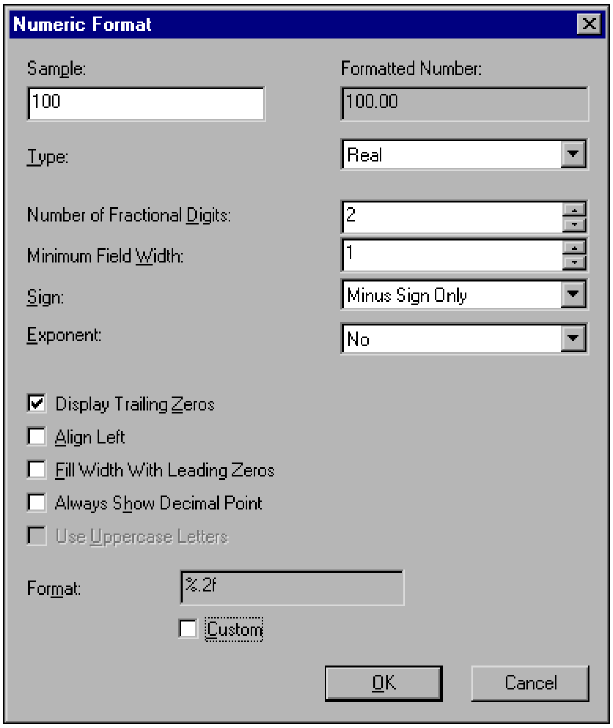
Measurement Must Be < 100 <Default>

Units: MHz

Figure 6-28. Edit Numeric Limits Test Dialog Box

25. Click on the **Numeric Format** button to open the Numeric Format dialog box. Set the control values of this dialog box to those shown in Figure 6-29. These settings specify the format of the step measurement and limit values. The format applies to the limit values that appear in the Edit Numeric Limit Test dialog box, the step description, and the test report.

    With these settings, TestStand compares the numeric measurement value that the VI returns to the constant value of 100. If the comparison is `True`, the step passes; otherwise, the step fails.


Numeric Format

Sample: 100

Formatted Number: 100.00

Type: Real

Number of Fractional Digits: 2

Minimum Field Width: 1

Sign: Minus Sign Only

Exponent: No

[✓] Display Trailing Zeros

[ ] Align Left

[ ] Fill Width With Leading Zeros

[ ] Always Show Decimal Point

[ ] Use Uppercase Letters

Format: %.2f

[ ] Custom

Figure 6-29. Numeric Format Dialog Box

26. Click **OK** twice to close the Numeric Format and Edit Numeric Limit Test dialog boxes.
27. Save the sequence as `Sample8.seq` by selecting **File»Save As** and save to the `<TestStand>\Tutorial` directory.

Building a LabWindows/CVI DLL

1. You now need to create a LabWindows/CVI project with which to build a DLL module. Launch LabWindows/CVI by selecting **Start»Programs»National Instruments»Measurement Studio» CVI IDE**.
2. Open a new project by selecting **File»New»Project**. If LabWindows/CVI already has a project loaded, it prompts you as to whether you want to unload the current project. Click the **Yes** button. LabWindows/CVI prompts you as to whether to transfer the current project options to the new project. Uncheck all options in the Transfer Project Options dialog box and click the **OK** button.

3. Select **Edit»Add Files to Project»All Files (*.*)** to open the Add Files to Project dialog box. Browse to each of the files below and use the Add button to add them to the Selected Files control.
   - `TestStand\Tutorial\FlexFrequency.c`
   - `TestStand\Tutorial\Frequency.uir`
   - `TestStand\API\CVI\tsapicvi.fp`
   - `TestStand\API\CVI\tsutil.fp`

   After adding all files, click the **OK** button to return to the project window where the file names should be displayed.
4. Save the project as `FlexFreuency.prj` in the `<TestStand>\Tutorial` directory by selecting **File»Save**.
5. Open the `FlexFequency.c` file by double clicking on the name of the file in the Project window.
6. Update the `FlexFrequency.c` code to prompt an operator to enter values into a frequency numeric control and a report text string control using the `frequency.uir` user interface resource file, as follows.
   a. Leaving the `FlexFrequency.c` window open, return to the project window by selecting **Windows»Project**.
   b. Open the `frequency.uir` file from the Project window.
   c. Right-click on the **Return** button and select **Generate Control Callback** from the context menu. If for some reason you have more than one `.c` file open, LabWindows/CVI prompts you to select a target file into which to insert the callback. Select `FlexFrequency.c` as the target.
   d. You just created the ReturnCallback function in `FlexFrequency.c` that quits the user interface. Close the `.uir` file after you are done since you will not be making any changes to it.
   e. Update the source code for the `ClockFrequency` and `ReturnCallback` function. The other functions in the source file do not need modification. Changed lines appear in bold.

```
void __declspec(dllexport) __stdcall
    ClockFrequency(CAObjHandle seqContextCVI, double
    *measurement, char addReportTxt[1024], short
    *errorOccurred, long *errorCode, char
    errorMsg[1024])
{
    int error = 0;
    int panelHandle, panel, control;
```

```
        panelHandle = LoadPanelEx (0, "frequency.uir",
            PANEL, __CVIUserHInst);

        if (panelHandle < 0)
            {
            error = panelHandle;
            goto Error;
            }

        DisplayPanel (panelHandle);

        // Automatically closes user display dialog when
        // TestStand execution terminates or aborts.
        TS_CancelDialogIfExecutionStops (panelHandle,
            seqContextCVI);

        RunUserInterface();

        // Assign values from UIR to return data
        GetCtrlVal (panelHandle, PANEL_FREQUENCY,
            measurement);
        GetCtrlVal (panelHandle,
            PANEL_ADDITIONAL_REPORT, addReportTxt);

    Error:
        // FREE RESOURCES
        DiscardPanel(panelHandle);

        // If an error occurred, set the error flag to
        cause
        // a run-time error in TestStand.
        if (error < 0)
        {
            *errorOccurred = TRUE;

        // OPTIONALLY SET THE ERROR CODE AND STRING
        *errorCode = error;
        strcpy(errorMsg, "A run-time error occurred.");
    }
    return;
}
```

```
int CVICALLBACK ReturnCallback (int panelHandle, int
    control, int event,void *callbackData, int
    eventData1, int eventData2)
{
    switch (event)
    {
        case EVENT_COMMIT:
            QuitUserInterface (0);
        break;
    }
    return 0;
}
```

7. From the Project window select **Build»Target Type»Dynamic Link Library** so that the project builds a DLL.
8. Build the DLL by selecting **Build»Create Debuggable Dynamic Link Library**. If the only option is **Build»Create Release Dynamic Link Library** then select **Build»Configuration»Debug** before building the DLL. LabWindows/CVI reports when the DLL is successfully built.
9. You are now ready to execute the sequence that calls your DLL function. Return to the sequence editor and execute the sequence by selecting **Execute»Single Pass**.
10. Click on **Done** in the Test Simulator dialog box. When TestStand executes the Clock Frequency Test step, the code module displays the user interface panel and waits for input.
11. Type a numeric value of 20 in the Frequency Measurement control.
12. Type any text in the Additional Report Text control.
13. Click on the **Return** button to continue the sequence execution.
14. When the sequence completes the execution, examine the test report. Notice the values for the status, measurement, and report text for the Clock Frequency step
15. Close the Execution window.

Debugging the DLL Function

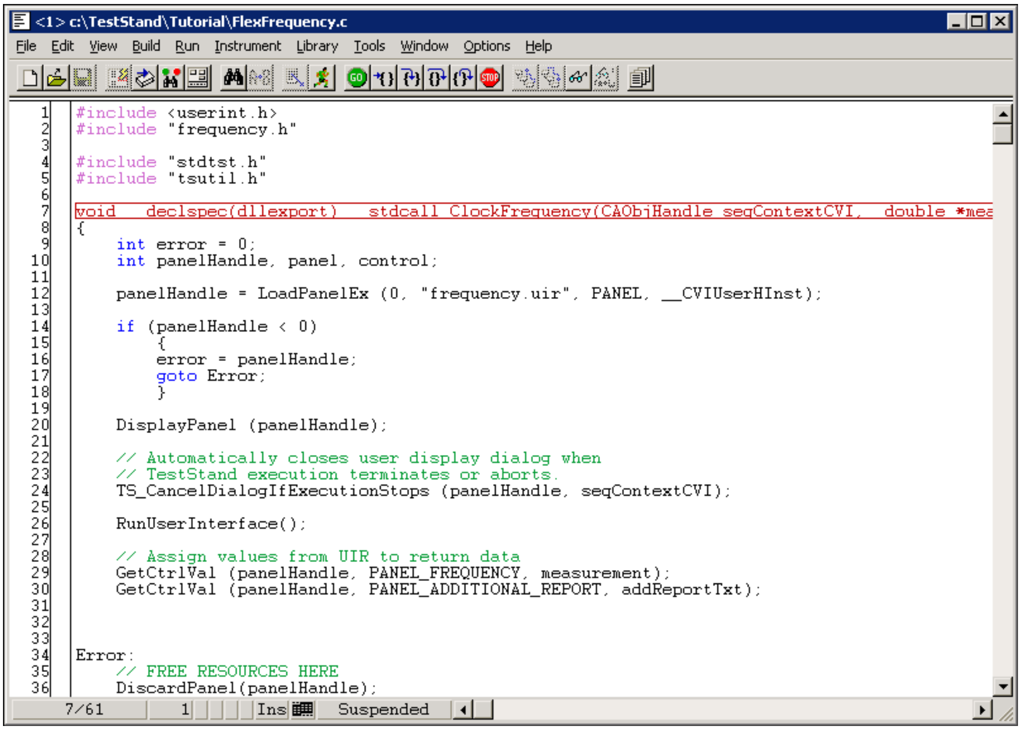
To debug DLLs that are called with the DLL Flexible Prototype Adapter, you must use the debugging features of the application development environment in which the DLL was written. This exercise shows how to use the capabilities of LabWindows/CVI to debug LabWindows/CVI DLL functions that are called using the DLL Flexible Prototype Adapter.

1. Exit the sequence editor.
2. In LabWindows/CVI, select **File»Open»Project**. Navigate to `<TestStand>\Tutorial\` and open `FlexFequency.prj`.
3. In the project window, select **Run»Select External Process**. Click the **Browse** button and select `<TestStand>\bin\SeqEdit.exe`.
4. Click **OK** to return to the LabWindows/CVI Project window. Select **Run»Run SeqEdit.exe** to launch the sequence editor from LabWindows/CVI. If prompted to save changes, select **Yes**. LabWindows/CVI starts the sequence editor as an external process and attaches to it for debugging.
5. Once the sequence editor has been launched and you have logged in, open your sequence file, `<TestStand>\Tutorial\Sample8.seq`. You also can find this file in the `<TestStand>\Tutorial\Solution` directory.
6. Place a breakpoint on the Clock Frequency Test step.

Note Place a breakpoint by clicking to the left of a step icon or by pressing F9 while your cursor is on that particular line.

7. Execute the sequence by selecting **Execute»Single Pass**.
8. Click on **Done** in the Test Simulator dialog box. When TestStand stops at the Clock Frequency Test step, press the F8 key to step into the DLL function. You can also step into the function by selecting **Debug»Step Into** or by using the Step Into button on the sequence editor toolbar.

   The execution stops on the first line of the function definition in `FlexFrequency.c` as shown in Figure 6-30. In the LabWindows/CVI debugging window, you can use the LabWindows/CVI stepping tools, watch window, and variable window to debug the code.


<1> c:\TestStand\Tutorial\FlexFrequency.c
File Edit View Build Run Instrument Library Tools Window Options Help
7/61 1 Ins Suspended

```
#include <userint.h>
#include "frequency.h"

#include "stdtst.h"
#include "tsutil.h"

void __declspec(dllexport) __stdcall ClockFrequency(CAObjHandle seqContextCVI, double *mea
{
    int error = 0;
    int panelHandle, panel, control;

    panelHandle = LoadPanelEx (0, "frequency.uir", PANEL, __CVIUserHInst);

    if (panelHandle < 0)
        {
        error = panelHandle;
        goto Error;
        }

    DisplayPanel (panelHandle);

    // Automatically closes user display dialog when
    // TestStand execution terminates or aborts.
    TS_CancelDialogIfExecutionStops (panelHandle, seqContextCVI);

    RunUserInterface();

    // Assign values from UIR to return data
    GetCtrlVal (panelHandle, PANEL_FREQUENCY, measurement);
    GetCtrlVal (panelHandle, PANEL_ADDITIONAL_REPORT, addReportTxt);



Error:
    // FREE RESOURCES HERE
    DiscardPanel(panelHandle);
```

Figure 6-30. Debugging the ClockFrequency function

9. The `ClockFrequency` function calls the `RunUserInterface` function, which displays the Clock Frequency dialog box. Because the LabWindows/CVI debugging window remains in the foreground while you are debugging, you may need to switch to the Clock Frequency dialog box to enter values in its controls.
10. Type a numeric value of 20 in the Frequency Measurement control of the Clock Frequency dialog box.
11. Type any text in the Additional Report Text control of the Clock Frequency dialog box.
12. Click on the **Return** button to continue.
13. LabWindows/CVI honors any breakpoints you set in the source files for the DLL. When the Clock Frequency step executes, the sequence execution stops at the first breakpoint you set in the source code.

14. Finish stepping through your DLL function until you return to the sequence execution and complete execution. This debugging technique can also be applied to steps that use the C/CVI Standard Prototype Adapter.
15. Close the sequence execution window and shut down the sequence editor.

Note You can use a similar technique in other application development environments from which you can launch and attach to an external process. For additional debugging information refer to National Instruments developer zone at `ni.com`.

This concludes this tutorial session. In the next session, you learn how to use the TestStand operator interfaces.

7

Using Run-Time Operator Interfaces

In this chapter, you learn how to use the LabWindows/CVI operator interface. The features that this chapter discusses also apply to the LabVIEW, Visual Basic, and Delphi operator interfaces. Refer to Chapter 16, *Run-Time Operator Interfaces* in the *TestStand User Manual* for more information on how to customize a run-time operator interface

TestStand includes run-time operator interfaces in both source and executable form. Each run-time operator interface is a separate application program. The operator interfaces differ primarily based on the language and Application Development Environment (ADE) in which each is developed. TestStand includes run-time operator interfaces developed in LabVIEW, LabWindows/CVI, Visual Basic, and Delphi. The TestStand run-time operator interfaces are less complex than the sequence editor and are fully customizable.

Loading Sequences

Complete the following steps to load a sequence in a run-time operator interface:

Note You can use any TestStand operator interface for this tutorial session. If you use the LabVIEW operator interface and you are using a version of LabVIEW more recent than 5.1, you should first mass compile the VIs in the TestStand installation directory. The screen shots in this session show the LabWindows/CVI operator interface.

1. Launch the LabWindows/CVI Operator interface from the Windows taskbar by selecting **Start»Programs»National Instruments TestStand»Operator Interfaces»LabWindows-CVI**.

   After the main window for the operator interface displays, a Login dialog box appears.
2. Select the `administrator` user name, leaving the password empty, and click on the **OK** button.

After you login, the operator interface appears as shown in Figure 7-1.

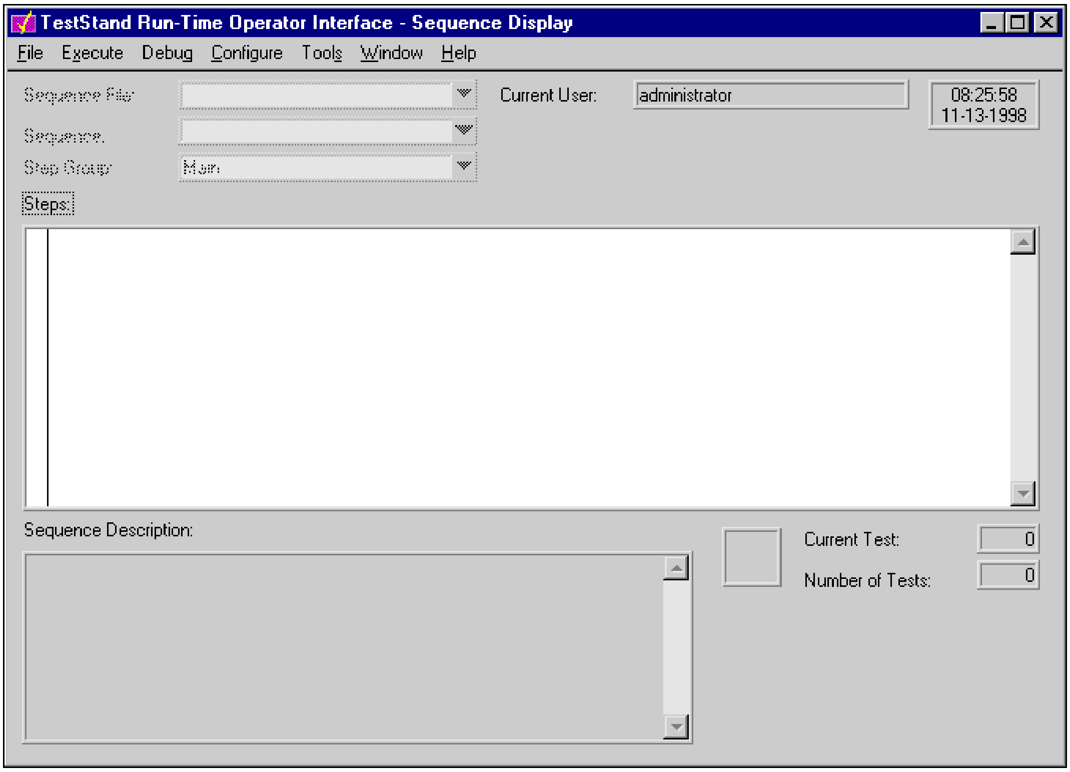


Figure 7-1. LabWindows/CVI Operator Interface

3. Using the mouse, browse through the menu options and notice that the operator interface menus contain many of the commands available from the sequence editor.

   Like the sequence editor, the run-time operator interfaces allow you start multiple concurrent executions, set breakpoints, and single-step. Unlike the sequence editor, however, the run-time operator interfaces do not allow you to modify sequences, and they do not display sequence variables, sequence parameters, step properties, and so on.

4. For this exercise, you will not be debugging code modules in LabWindows/CVI. Verify that the LabWindows/CVI Standard Prototype Adapter is properly configured to execute code modules in the same process as the operator interface, as follows:

   a. Select **Configure»Adapters**, which displays the Adapter Configuration dialog box.

b. Select the C/CVI Standard Prototype Adapter in the Configurable Adapters section.

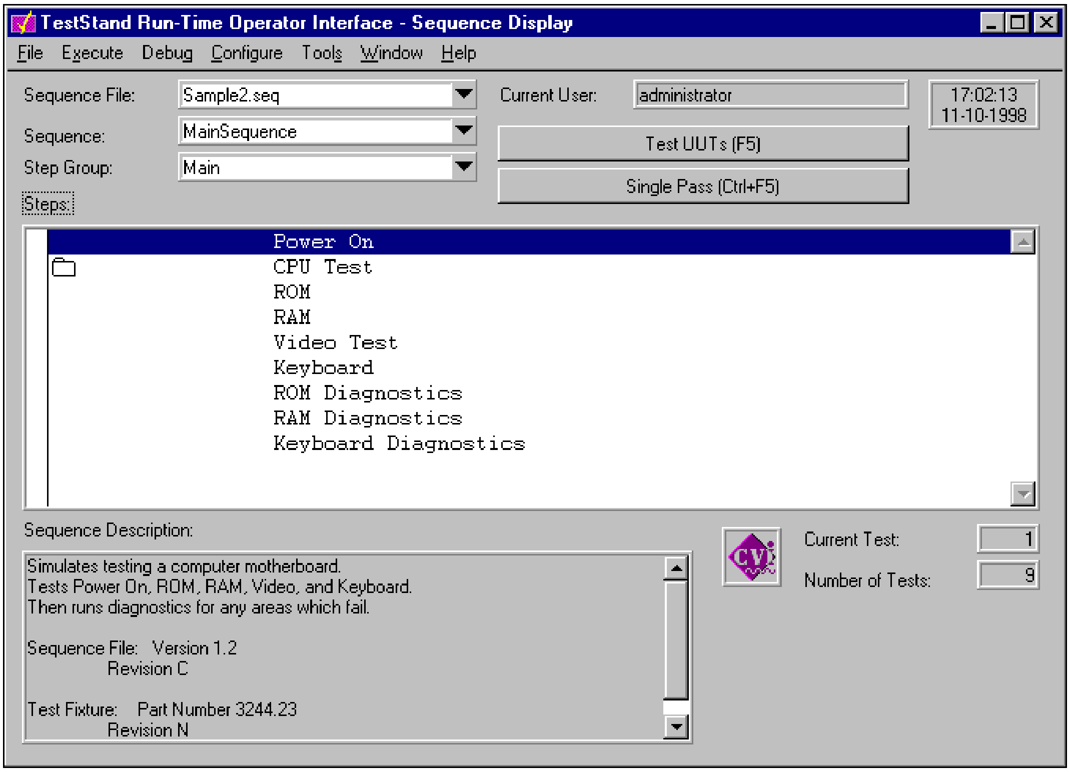
c. Click on the **Configure** button, which displays the CVI Standard Adapter Configuration dialog box.

d. Enable the Execute Steps In-Process option.

e. Click on **OK**, then click on **Done** to close the configuration dialog boxes.

5. Select **File»Open Sequence File** and open the file `<TestStand>\Tutorial\Sample2.seq`, which you created in Chapter 3, *Editing Steps in a Sequence*. You also can find this file in the `<TestStand>\Tutorial\Solution` directory.

   After you open the sequence file, the operator interface window appears as shown in Figure 7-2.


TestStand Run-Time Operator Interface - Sequence Display
File Execute Debug Configure Tools Window Help
Test UUTs (F5)
Single Pass (Ctrl+F5)
Steps:

Sequence File: Sample2.seq
Sequence: MainSequence
Step Group: Main
Current User: administrator
17:02:13
11-10-1998

Power On
CPU Test
ROM
RAM
Video Test
Keyboard
ROM Diagnostics
RAM Diagnostics
Keyboard Diagnostics

Sequence Description:

Simulates testing a computer motherboard.
Tests Power On, ROM, RAM, Video, and Keyboard.
Then runs diagnostics for any areas which fail.

Sequence File: Version 1.2
Revision C

Test Fixture: Part Number 3244.23
Revision N

Current Test: 1
Number of Tests: 9

Figure 7-2. Open Sequence in Operator Interface

The Sequence File, Sequence, and Step Group ring controls specify the steps that the operator interface displays in the Steps list box control.

6. Select the `Setup` step group by clicking on the arrow to the right of the right of the Step Group control to view its steps.
7. Reselect `Main` to return to the Main step group.

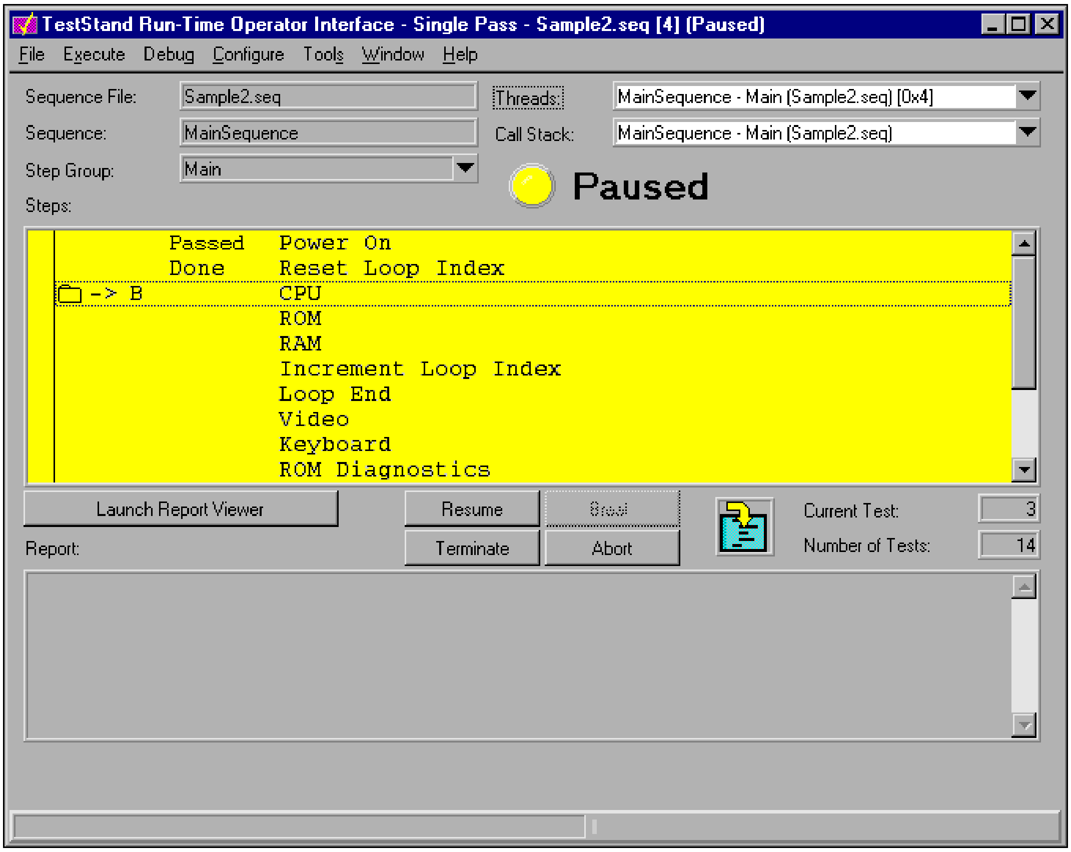
Running and Debugging Sequences

Complete the following steps to run and debug a sequence in a run-time operator interface:

1. Set a breakpoint on the CPU Test step by clicking on the step and then selecting **Debug»Toggle Breakpoint**. Notice that the letter “B” appears to the left of the step name.
2. Execute the sequence by selecting **Execute»Single Pass**.

   When you start the execution, the operator interface displays the execution in a separate window.

3. When the execution displays the Test Simulator dialog box, click on **Done**. The execution then pauses at the breakpoint on the CPU Test step, as shown in Figure 7-3.


TestStand Run-Time Operator Interface - Single Pass - Sample2.seq [4] (Paused)
File Execute Debug Configure Tools Window Help
Sequence File: Sample2.seq
Sequence: MainSequence
Step Group: Main
Steps:
Threads: MainSequence - Main (Sample2.seq) [0x4]
Call Stack: MainSequence - Main (Sample2.seq)
Paused
Launch Report Viewer
Report:
Resume
Break
Terminate
Abort
Current Test: 3
Number of Tests: 14

```
        Passed   Power On
        Done     Reset Loop Index
-> B             CPU
                 ROM
                 RAM
                 Increment Loop Index
                 Loop End
                 Video
                 Keyboard
                 ROM Diagnostics
```

Figure 7-3. Paused Execution in Operator Interface

4. Select **Debug»Step Into**. Notice that the Execution window changes and displays the steps in the `MainSequence` sequence of the `SubSequence1.seq` sequence file.
5. Single-step a few times by selecting **Debug»Step Over**.
6. Select **Debug»Resume** to complete the execution.

   All of the single-stepping and stepping into source code modules are available to you from the operator interface applications.
7. Close the Execution window by selecting **File»Close Execution**.

Running Multiple Executions

Complete the following steps to open a new sequence file and start multiple executions:

1. Select **File»Open Sequence File** and open the `LoopForever.seq` sequence file from the `<TestStand>\Tutorial` directory. This sequence contains a series of empty steps that continuously loop back to the first step.
2. Start an execution by selecting **Execute»Run "MainSequence"**.

   After the new Execution window appears, drag the Execution window off to the side so you can see the Sequence Display window.
3. Start a second execution by again selecting **Execute»Run "MainSequence"** in the Sequence Display window.

   If you do not see the **Run "MainSequence"** command in the menu, confirm that you have selected the correct window by clicking on its tab.
4. Create a total of four executions. Each of the executions you start executes under the main operator interface process.
5. To terminate all the executions you started, select **Debug»Terminate All**. The **Terminate All** command is available from any of the operator interface windows.
6. Close all the Execution windows by selecting **Window»Close All Completed Executions**.
7. Close the main operator interface window by selecting **File»Exit**.

This concludes this tutorial session. In the next session, you learn how to use callbacks.

8 Using Callbacks

In this chapter, you learn how to customize the execution of a sequence within TestStand using callbacks. *Callbacks* are sequences that are used to handle common tasks such as serial number inquiry or report logging.

All of the default callback sequences that TestStand includes are provided in source form so that you can edit or replace them to customize TestStand for your particular application. In this tutorial session, you replace one of the default TestStand callbacks with your own.

Setting Up the Example

If you did not directly proceed from Chapter 7, *Using Run-Time Operator Interfaces*, follow these steps to set up the TestStand sequence editor so you can complete this tutorial session:

1. If the sequence editor is not running, launch the sequence editor.
2. Close all windows in the sequence editor.

Overriding a Process Model Callback

The TestStand process models contain sequences that define operations TestStand performs before and after it tests a UUT. If you want to invoke a sequence in one of the process models, you can run one of the entry point sequences in the models. Each model uses the default model entry points `Test UUTs` and `Single Pass`, as discussed in Chapter 2, *Loading and Running Sequences*, of this manual. The process models contain hooks that allow you to customize the behavior of a process model for each main sequence that uses it without forcing you to edit the process model directly. These hooks are in the form of sequences and are called *model callbacks*.

For example, the TestStand process models define a `TestReport` callback that generates the test report for each UUT. Normally, the `TestReport` callback in the process model files is sufficient because it handles many types of test results. The sequence developer can, however, override the default `TestReport` callback by defining a different `TestReport` callback in a particular client sequence file. To alter the behavior of the process models for all sequences, you can modify the process models or replace the models entirely.

Execution entry points in process models use callbacks to invoke the main sequence in the client sequence file. Each client sequence file must define a sequence by the name of `MainSequence`. The process models contain a `MainSequence` callback that is merely a place holder. The `MainSequence` in the client sequence file overrides the `MainSequence` place holder in a model file.

Figure 8-1 shows the callbacks that the default TestStand process model, `SequentialModel.seq`, calls and the order in which TestStand executes the callbacks within the Test UUT execution entry point.

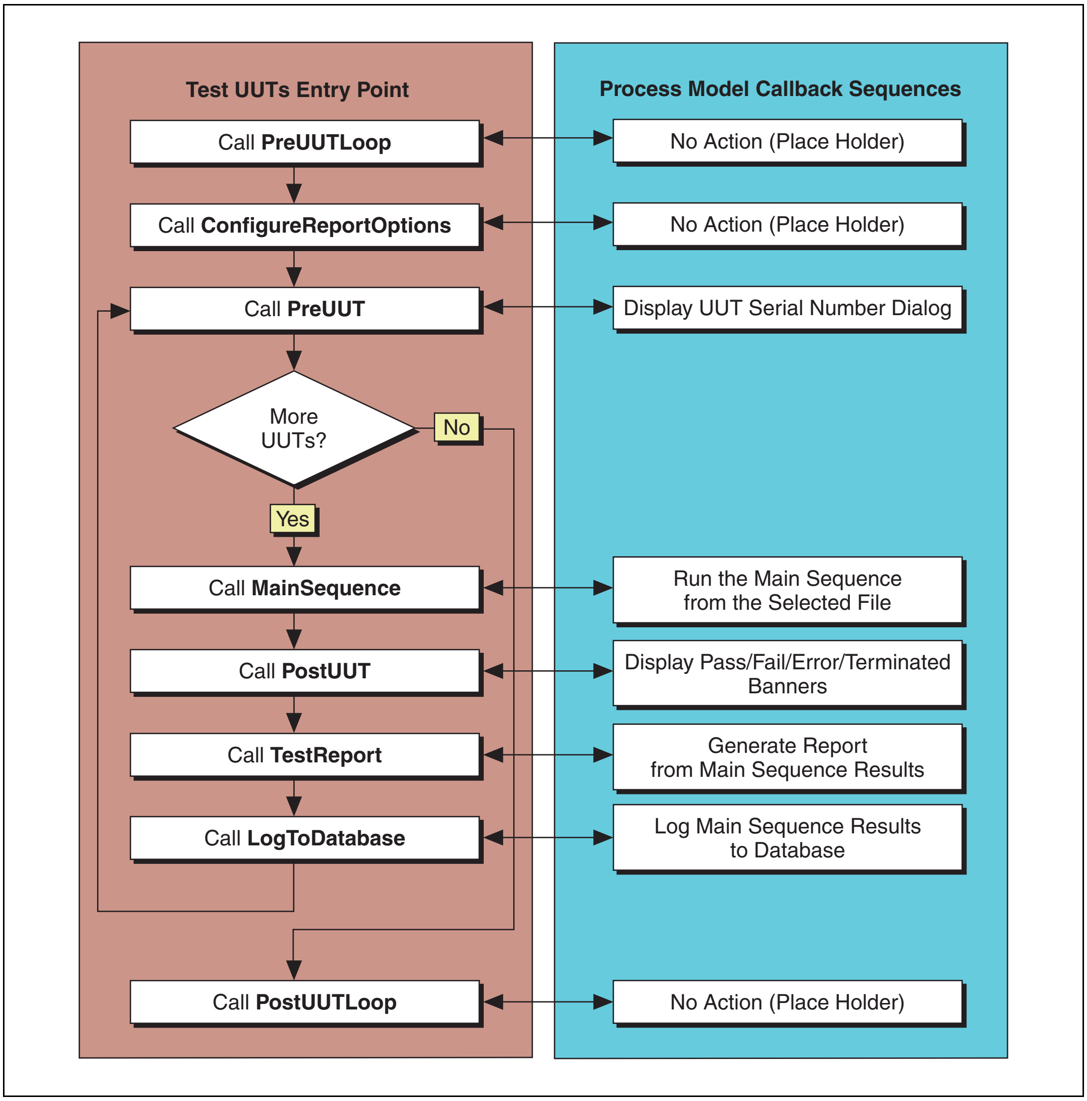
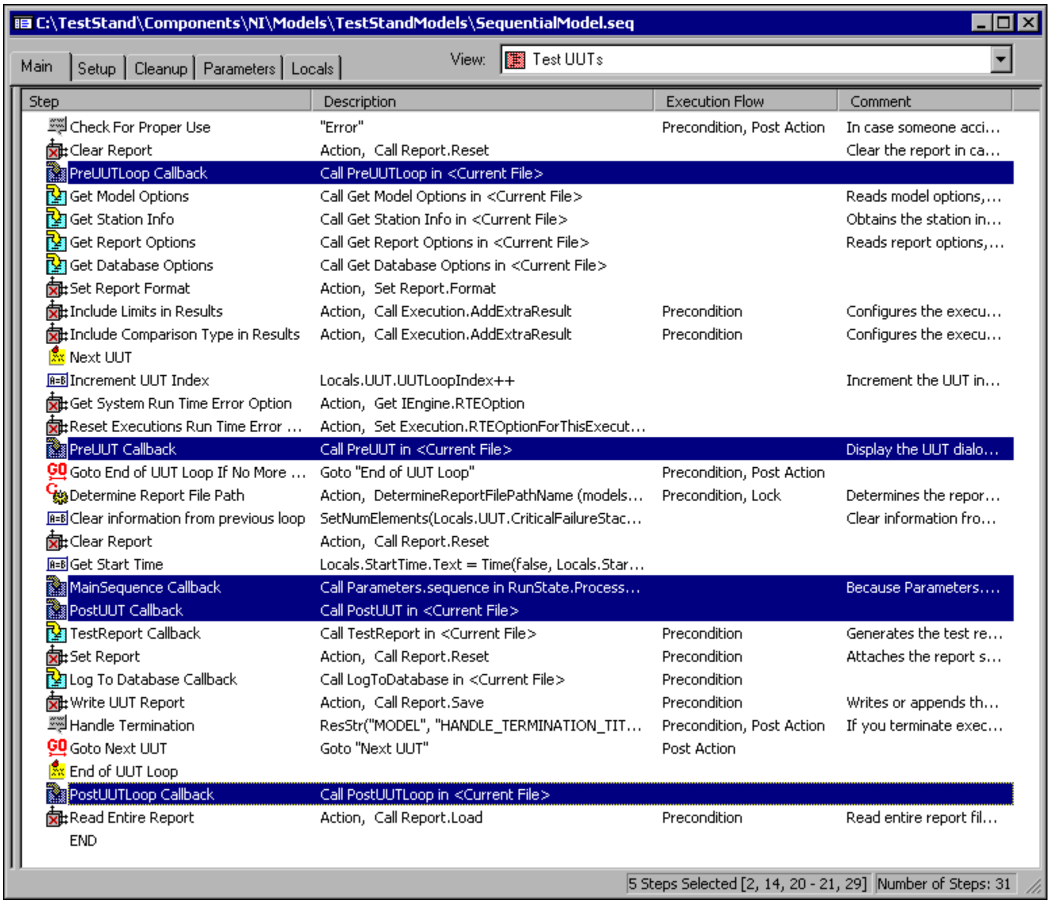

Test UUTs Entry Point
Process Model Callback Sequences
Call PreUUTLoop
No Action (Place Holder)
Call ConfigureReportOptions
No Action (Place Holder)
Call PreUUT
Display UUT Serial Number Dialog
More UUTs?
No
Yes
Call MainSequence
Run the Main Sequence from the Selected File
Call PostUUT
Display Pass/Fail/Error/Terminated Banners
Call TestReport
Generate Report from Main Sequence Results
Call LogToDatabase
Log Main Sequence Results to Database
Call PostUUTLoop
No Action (Place Holder)

Figure 8-1. TestStand Sequential Model Callbacks

Follow these steps to add a PreUUTLoop callback that displays a message popup prompt:

1. Open `<TestStand>\Components\NI\Models\TestStandModelSequentialModel.seq`. You also can find this file in the `<TestStand>\Tutorial\Solution` directory. This file is the default process model TestStand uses to execute sequences.
2. Select the sequence `TestUUTs` from the View selector ring. This sequence is the `Test UUTs` entry point that TestStand executes when you select **Execute»Test UUTs**. Notice the callback sequences that the `Test UUTs` sequence calls, such as `PreUUTLoop Callback`, `PreUUT Callback`, `MainSequence Callback`, and `PostUUTLoop Callback`, shown in Figure 8-2.


C:\TestStand\Components\NI\Models\TestStandModels\SequentialModel.seq

Main | Setup | Cleanup | Parameters | Locals

View: Test UUTs

Step	Description	Execution Flow	Comment
Check For Proper Use	"Error"	Precondition, Post Action	In case someone acci...
Clear Report	Action, Call Report.Reset		Clear the report in ca...
PreUUTLoop Callback	Call PreUUTLoop in <Current File>		
Get Model Options	Call Get Model Options in <Current File>		Reads model options,...
Get Station Info	Call Get Station Info in <Current File>		Obtains the station in...
Get Report Options	Call Get Report Options in <Current File>		Reads report options,...
Get Database Options	Call Get Database Options in <Current File>		
Set Report Format	Action, Set Report.Format		
Include Limits in Results	Action, Call Execution.AddExtraResult	Precondition	Configures the execu...
Include Comparison Type in Results	Action, Call Execution.AddExtraResult	Precondition	Configures the execu...
Next UUT			
Increment UUT Index	Locals.UUT.UUTLoopIndex++		Increment the UUT in...
Get System Run Time Error Option	Action, Get IEngine.RTEOption		
Reset Executions Run Time Error ...	Action, Set Execution.RTEOptionForThisExecut...		
PreUUT Callback	Call PreUUT in <Current File>		Display the UUT dialo...
Goto End of UUT Loop If No More ...	Goto "End of UUT Loop"	Precondition, Post Action	
Determine Report File Path	Action, DetermineReportFilePathName (models...	Precondition, Lock	Determines the repor...
Clear information from previous loop	SetNumElements(Locals.UUT.CriticalFailureStac...		Clear information fro...
Clear Report	Action, Call Report.Reset		
Get Start Time	Locals.StartTime.Text = Time(false, Locals.Star...		
MainSequence Callback	Call Parameters.sequence in RunState.Process...		Because Parameters....
PostUUT Callback	Call PostUUT in <Current File>		
TestReport Callback	Call TestReport in <Current File>	Precondition	Generates the test re...
Set Report	Action, Call Report.Reset	Precondition	Attaches the report s...
Log To Database Callback	Call LogToDatabase in <Current File>	Precondition	
Write UUT Report	Action, Call Report.Save	Precondition	Writes or appends th...
Handle Termination	ResStr("MODEL", "HANDLE_TERMINATION_TIT...	Precondition, Post Action	If you terminate exec...
Goto Next UUT	Goto "Next UUT"	Post Action	
End of UUT Loop			
PostUUTLoop Callback	Call PostUUTLoop in <Current File>		
Read Entire Report	Action, Call Report.Load	Precondition	Read entire report fil...
END			

5 Steps Selected [2, 14, 20 - 21, 29] | Number of Steps: 31

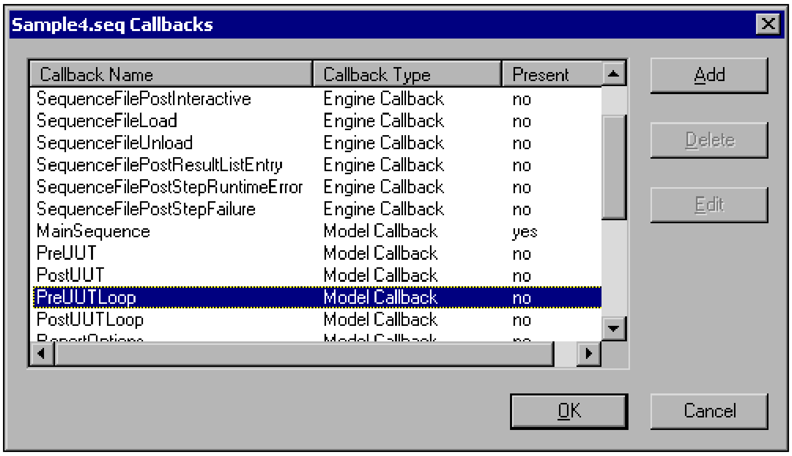
Figure 8-2. Test UUTs Sequence

3. Right click on the PreUUT Callback step and select **Open Sequence** from the context menu. The PreUUT Callback sequence has two steps: Identify UUT and Set Serial Number.
4. Right click on the Identify UUT step and select **Run Selected Steps** from the context menu. A familiar-looking dialog box should appear.

   This step displays the UUT Information dialog box when you execute a sequence using the Test UUT entry point. If you wanted to change the way in which TestStand obtained a UUT serial number, such as reading it from a bar code, you would replace this callback with your own.
5. Click on **OK** on the UUT Information dialog box.
6. Close the execution window.
7. Select `Test UUTs` again from the View selector ring.
8. Right click on the PreUUTLoop Callback step and select **Open Sequence** from the context menu. Notice that this callback is empty. The empty sequence is a place holder, so that if you want to add steps that execute before the UUT loop, you can create them in this callback.
9. Close the `SequentialModel.seq` sequence file window, and do not save any changes if the sequence editor prompts you to.

   You override the PreUUTLoop Callback step with your own callback sequence.
10. Open the file `<TestStand>\Tutorial\Sample4.seq`, which you created in Chapter 5, *Using Variables and Properties*. You also can find this file in the `<TestStand>\Tutorial\Solution` directory.

11. Select **Edit»Sequence File Callbacks** to display the `Sample4.seq` Callbacks dialog box, shown in Figure 8-3.



Callback Name	Callback Type	Present
SequenceFilePostInteractive	Engine Callback	no
SequenceFileLoad	Engine Callback	no
SequenceFileUnload	Engine Callback	no
SequenceFilePostResultListEntry	Engine Callback	no
SequenceFilePostStepRuntimeError	Engine Callback	no
SequenceFilePostStepFailure	Engine Callback	no
MainSequence	Model Callback	yes
PreUUT	Model Callback	no
PostUUT	Model Callback	no
PreUUTLoop	Model Callback	no
PostUUTLoop	Model Callback	no

Figure 8-3. Adding Callbacks to a Sequence

12. Select the `PreUUTLoop` callback name.
13. Click on the **Add** button.

    Notice that the value in the Present column changes from no to yes. When clicking on the **Add** button, the sequence editor creates a new empty callback sequence to your sequence file. Now, when you start an execution using a model entry point, TestStand calls the callback in your sequence file instead of the sequence in the `SequentialModel.seq` process model.
14. Click on **OK** to close the callbacks dialog box.
15. Select All Sequences in the View selector ring of the sequence file window. Notice that the sequence file now contains two sequences: `MainSequence` and `PreUUTLoop`.
16. Right click on the `PreUUTLoop` sequence and select **View Contents** from the context menu.
17. Right click inside the step list pane of the Main tab and select **Insert Step»Message Popup** from the context menu.
18. Rename the new step `Pre UUT Message`.
19. Right click on the Pre UUT Message step and select **Edit Message Settings** from the context menu.

20. In the Title Expression control, enter the literal string `"Pre UUT Loop Callback Message"`. You must enclose string literals you enter in any expression field in double quotation marks (`"`).
21. In the Message Expression, enter the literal string `"Now in the Pre UUT Loop Callback"`.
22. Click on **OK** to close the Edit Message Settings dialog box.
23. Save the sequence by selecting **File»Save As**. Save the sequence as `Sample9.seq` in the `TestStand\Tutorial` directory.
24. Execute the sequence by selecting **Execute»Test UUTs**. Notice that the Pre UUT Loop Callback Message dialog box is the first prompt TestStand displays.
25. Click on **OK** to close the dialog box. TestStand now displays the UUT Information dialog box from the `PreUUT Callback` sequence in the `SequentialModel.seq` process model.
26. Enter a serial number and click on **OK**.
27. Run through several iterations of the sequence.
28. Click on **Stop** in the UUT Information dialog box.

    Notice that TestStand displays only the Pre UUT Loop Callback Message dialog box once at the very beginning of the execution. The reason is that, as seen in Figure 8-1, the `PreUUTLoop Callback` is executed before the loop, while the `PreUUT Callback` is executed within the loop.
29. Close all windows in the sequence editor.

You can make modifications similar to those in this example to the other TestStand process models, `ParallelModel.seq` and `BatchModel.seq`. These models are discussed in Chapter 2, *Loading and Running Sequences*. For more information about the process models, callbacks, and modifying callbacks, refer to Chapter 1, *TestStand Architecture Overview*, Chapter 3, *Configuring and Customizing TestStand*, and Chapter 14, *Process Models*, in the *TestStand User Manual*.

This concludes this session of the tutorial. In the next session, you learn how to add users and configure user privileges in TestStand.

9 Adding Users and Setting Privileges

This chapter discusses how to use the TestStand User Manager, and how you can add new users and change their privileges.

Setting Up the Example

If you did not directly proceed from Chapter 8, *Using Callbacks*, close all windows in the sequence editor so you can complete this tutorial session.

Using the User Manager

The TestStand Sequence Editor includes a User Manager for adding and removing users, and for managing the privileges of each user.

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

In this exercise, you learn how to view current users and add new ones.

1. Launch the User Manager window by selecting **View»User Manager**. The left pane shows all of the users configured on this station. Expand the `administrator` tree node to display the hierarchy of properties associated with this user. The `Privileges` node contains settings for all actions a user can perform, such as executing sequences, debugging sequences, or adding new users.

   When you highlight a node in the tree view, the corresponding property values under the node appear in the list pane on the right. Notice that all the property values for the privileges under the administrator user are `True`.

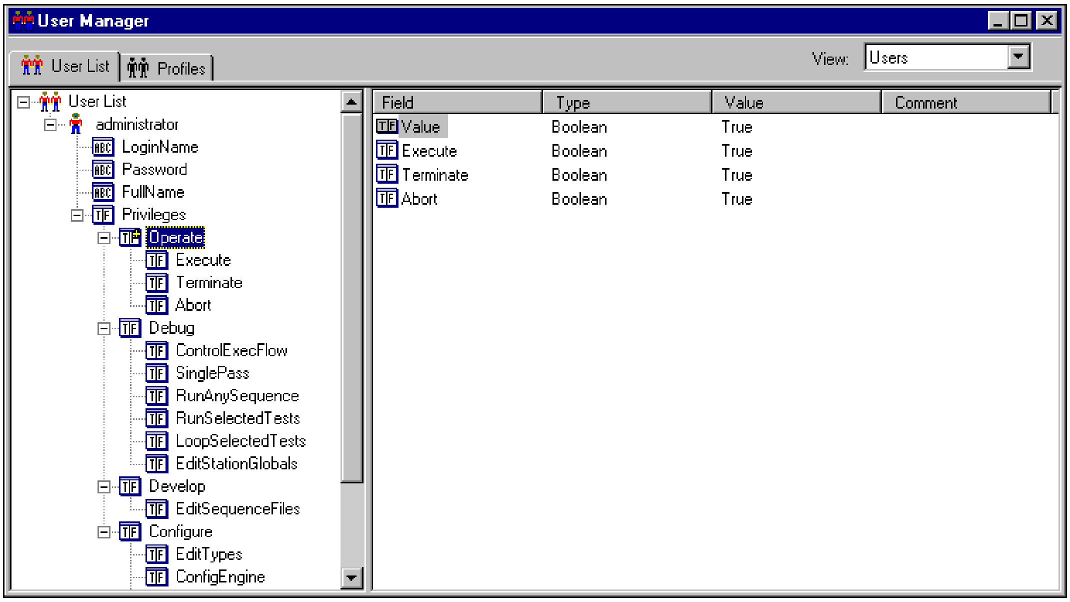
The privileges are organized in hierarchical groups. Each privilege group has a Boolean subproperty named `GrantAll`. A user has a privilege if you set the property of the privilege to `True`. Alternatively, you can set the `GrantAll` property of a privilege group to specify whether a user has all privileges within a privilege group, regardless of the property value of the individual privileges.

Note The property `User.Privileges.GrantAll` applies to all privilege groups. If this property is set to `True`, the user has all privileges. This property must be set to `False` to honor privilege settings within each privilege group.

You can grant privileges in several different ways. The following example demonstrates one way that a privilege can be granted. A user has the privilege to terminate an execution if one of the following set of conditions is met.

- `User.Privileges.GrantAll` is set to `True`. This also grants rights to all other privileges.
- `User.Privileges.GrantAll` is set to `False` and `User.Privileges.Operate.GrantAll` is set to `True`. This also grants rights to other privileges of the Operate privilege group.
- `User.Privileges.GrantAll` and `User.Privileges.Operate.GrantAll` are set to `False` and `User.Privilege.Operate.Terminate` is set to `True`. This only ensures that a user has the privilege to terminate an execution.

Figure 9-1 shows the expanded tree view for the properties under the `administrator` user.



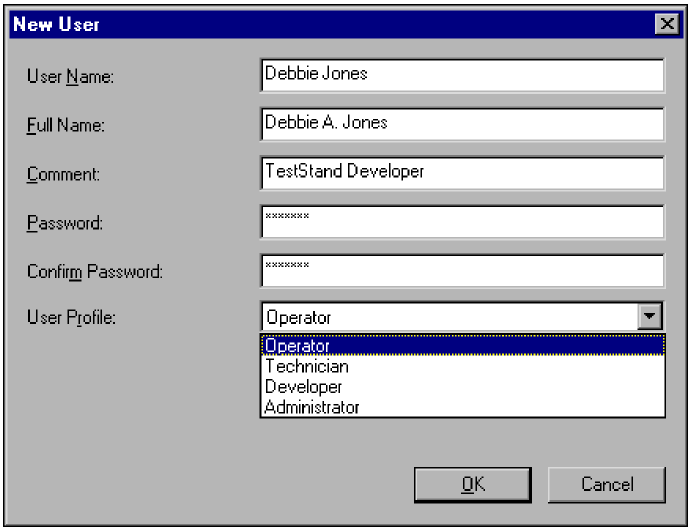
- User List
  - administrator
    - LoginName
    - Password
    - FullName
    - Privileges
      - Operate
        - Execute
        - Terminate
        - Abort
      - Debug
        - ControlExecFlow
        - SinglePass
        - RunAnySequence
        - RunSelectedTests
        - LoopSelectedTests
        - EditStationGlobals
      - Develop
        - EditSequenceFiles
      - Configure
        - EditTypes
        - ConfigEngine

Field	Type	Value	Comment
Value	Boolean	True	
Execute	Boolean	True	
Terminate	Boolean	True	
Abort	Boolean	True	

Figure 9-1. User Manager Window

2. Add a new user as follows:
   a. Click on the User List node in the tree view, which lists the administrator user in the right pane.
   b. Right click in the right pane and select **Insert User** from the context menu, which displays the New User dialog box.
   c. Fill in the User Name and Full Name controls with your name.
   d. Type your password into the Password and Confirm Password controls.

Figure 9-2 shows an example of a completed New User dialog box.



User Name:	Debbie Jones
Full Name:	Debbie A. Jones
Comment:	TestStand Developer
Password:	*******
Confirm Password:	*******
User Profile:	Operator

Operator
Technician
Developer
Administrator

Figure 9-2. New User Dialog Box

e. For the User Profile control, select `Operator`.

   User profiles define an initial set of privilege settings to give the new user. By default, the `Operator` profile grants a user the privilege to execute, terminate and abort sequences, but does not grant the privilege to create or debug sequences. TestStand provides four user profiles by default: `Operator`, `Technician`, `Developer`, and `Administrator`.

f. Click on **OK** to close the New User dialog box.

In addition to adding new users, the User Manager allows you to modify the default profiles and to create new profiles that define a combination of privileges appropriate for your test station.

3. Create a new profile as follows:

   a. Click on the Profiles tab of the User Manager window. In the list pane you should see the four default profiles.

   b. Click on the Profiles node in the tree view, which lists the currently defined profiles in the right list pane.

   c. Right click on the `Operator` profile in the right pane and select **Copy** from the context menu.

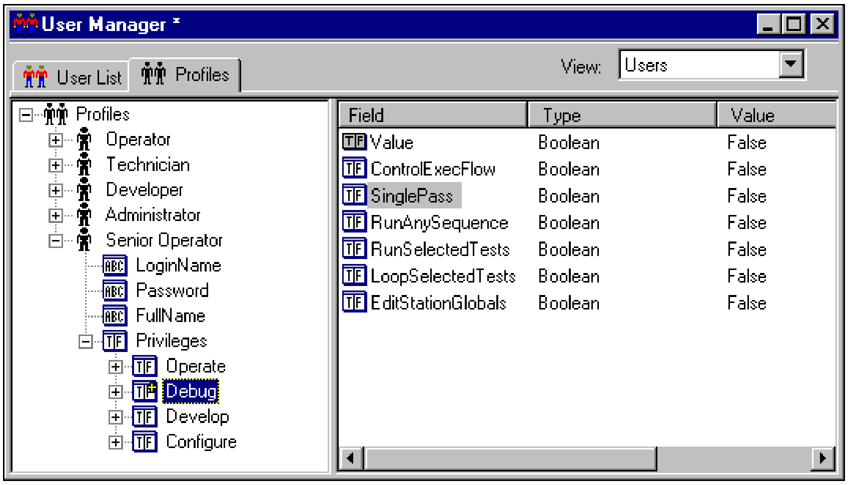
   d. Right click in the right pane and select **Paste**.

e. Rename the new profile `Senior Operator`. The new profile is identical to the `Operator` profile.

Note If the `Senior Operator` profile already exists, the paste operation appends an underscore and a unique number on the end of the name.

If you make changes to the values in a profile, your changes do not affect the privileges for users that already exist in the user list. After you create a user, you must modify privileges individually. You cannot modify privileges for existing users by changing user profiles.

4. Modify the default privileges for this new profile, as follows:
   a. Select the new Senior Operator node in the tree view and expand its privilege settings.
   b. Select the Debug node in the tree view, as shown in Figure 9-3.



Field	Type	Value
Value	Boolean	False
ControlExecFlow	Boolean	False
SinglePass	Boolean	False
RunAnySequence	Boolean	False
RunSelectedTests	Boolean	False
LoopSelectedTests	Boolean	False
EditStationGlobals	Boolean	False

Figure 9-3. Configure Privileges in New Profile

`Debug` is a Container property, which contains Boolean subproperties. The values all the subproperties under `Debug` are `False`.

You can set the value of each privilege in the right pane by right clicking on an item and selecting **Properties** from the context menu.

You can override the privileges in a privilege group and grant a user access to all privileges in a group by setting the value of the `GrantAll` property in the group to `True`.

Set the `SinglePass` property under the `Debug` group to `True` for the `Senior Operator` profile as follows:

   a. Double-click on the Single Pass name in the right pane. The Boolean Properties dialog box appears.
   b. Change the value to `True`.
   c. Click on **OK** to close the dialog box.

5. Add a new user using the `Senior Operator` profile, as follows:
   a. Click on the User List tab.
   b. Right click in the right pane and select **Insert User** from the context menu.
   c. Enter the information in the New User dialog box as in the previous step, this time using a different user name.
   d. Select the `Senior Operator` profile.
   e. Click on **OK** to close the New User dialog box.
6. Select **File»Login**. The two new users you just added now appear in addition to the `administrator` user name.
7. Select the user you created with the `Senior Operator` profile.
8. Enter the appropriate password.
9. Click on **OK**.
10. Open `Sample1.seq` from the `TestStand\Tutorial` Directory.
11. Open the **Execute** menu. Notice that the **Single Pass** menu is selectable, but the command **Run MainSequence** is grayed out because you no longer have the privilege to execute sequences without a model entry point.
12. Try to right click in a sequence view to insert a new step, and notice that the insert menu command is also grayed out because your privileges have changed.
13. Close all of the windows in the sequence editor.
14. Select **File»Login**.
15. Log in as the `administrator`. The password for the Administrator is an empty string.

You can add and remove users programatically using the TestStand API. The shipping example, `<TestStand>\Example\CreateDeleteUsers\CreateDeleteUsers.seq`, demonstrates how to add and remove users.

This concludes this session of the tutorial. In the next session, you learn how to use the TestStand ActiveX API in code modules.

10 Using ActiveX in Code Modules

This chapter teaches you how to use ActiveX from within a code module in TestStand. If you do not use LabVIEW or LabWindows/CVI, you can skip this chapter and proceed to Chapter 11, *Additional Development Features*.

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

As discussed in Chapter 5, *Using Variables and Properties*, variables are properties that you can freely create in certain contexts. You can have variables that are *global* to a sequence file or *local* to a particular sequence. You can also have *station global* variables. The values of station global variables are persistent across different executions and even across different invocations of the sequence editor or run-time operator interfaces. The TestStand engine maintains the value of station global variables in a file on the run-time computer.

Each step in a sequence can have properties. For example, a step might have an integer error code property. The type of a step determines the set of properties it has. For example, the Numeric Limits Test step contains properties for the comparison type and the high and low limit values.

You can use TestStand variables to share data among tests you write in different programming languages even if they do not have compatible data representations. You can pass values that you store in variables and properties to code modules. Also, you can use the TestStand ActiveX API to access variable and property values directly from code modules. When executing sequences, TestStand maintains a *sequence context* that contains references to all global variables, all local variables, and step properties in active sequences. The contents of the sequence context change depending on the currently executing sequence and step. If you pass a sequence context object reference to the code module, you can use the TestStand ActiveX API to access the variables and properties in the sequence context.

In this session, you learn how create code modules in the LabVIEW and LabWindows/CVI development environments that use the ActiveX API to share data with TestStand.

Using ActiveX in LabVIEW Test Virtual Instruments

In this exercise, you create a new sequence with two steps that call a LabVIEW VI. The first step generates an array of data and stores it in a TestStand variable. The second step plots the data stored in the variable. This session of the tutorial assumes a general familiarity with the LabVIEW development environment. If you are not using LabVIEW, but you do use LabWindows/CVI, you can skip this section and proceed to the *Using ActiveX in LabWindows/CVI Code Modules* section in this chapter.

Note Verify that you are using the appropriate version of LabVIEW with TestStand. Refer to the `readme.txt` file in the `TestStand\Doc` directory for more details.

Setting Up the Example

Close all windows in the sequence editor so you can complete this tutorial session.

Creating the Sequence and Virtual Instrument Tests

In this exercise, you create a new sequence in the sequence editor.

1. Open a new sequence by selecting **File»New Sequence File** in the menu.
2. Save the sequence by selecting **File»Save As**. Save the sequence as `Sample10.seq` in the `<TestStand>\Tutorial` directory. By saving the sequence file now, you can specify relative paths to code modules instead of absolute paths.
3. Click on the Locals tab of the Sequence File window.

4. Right click in the right pane and insert a numeric array variable, as shown in Figure 10-1.

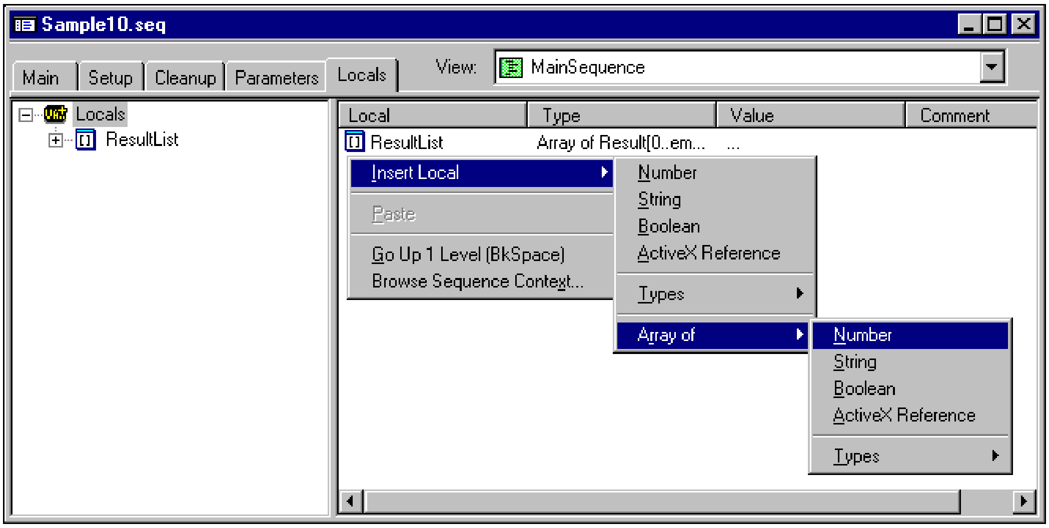
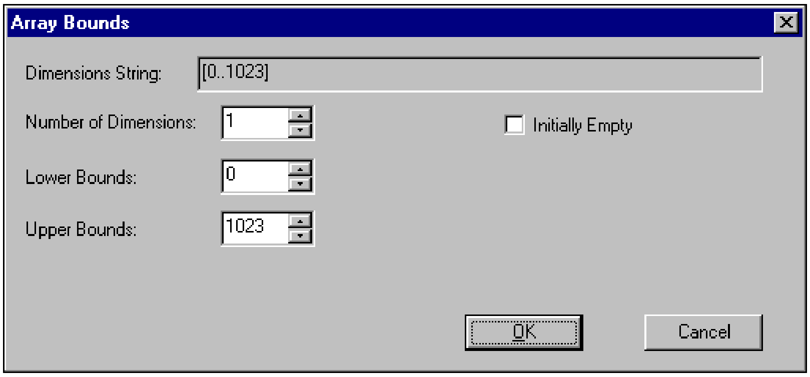


Figure 10-1. Insert Locals Array of Numeric

When you make this selection, the sequence editor displays the Array Bounds dialog box, shown in Figure 10-2.



Dimensions String: [0..1023]
Number of Dimensions: 1
Lower Bounds: 0
Upper Bounds: 1023

Figure 10-2. Array Bounds Dialog Box

5. Enter a value of `1023` for the Upper Bounds control as shown in Figure 10-2.
6. Click on **OK** to close the dialog box.
7. Right click on the new variable and select **Rename** from the context menu.
8. Rename the variable `Arraydata`.
9. Click on the Main tab of the Sequence File window. Select LabVIEW Standard Prototype Adapter in the Adapter Selector Ring control.
10. Right click in the step list and select **Insert Step»Action** to insert a new Action step.
11. Rename the step `Get Array Data From LabVIEW`.
12. Right click on the Get Array Data From LabVIEW step and select **Specify Module** in the context menu.
13. Using the **Browse** button on the Edit LabVIEW VI Call dialog box, select the file `GenerateWaveform.vi` in the `TestStand\Tutorial` directory.
14. On the Edit LabVIEW VI Call dialog box, enable the options Sequence Context ActiveX Pointer and Show VI Front Panel When Called.
15. Now that you have specified the VI, click on the **Edit Code** button on the Edit LabVIEW VI Call dialog box to open the VI in LabVIEW.

Figure 10-3 shows the front panel of the `GenerateWaveform.vi`. This VI generates a waveform based on the inputs from the front panel.

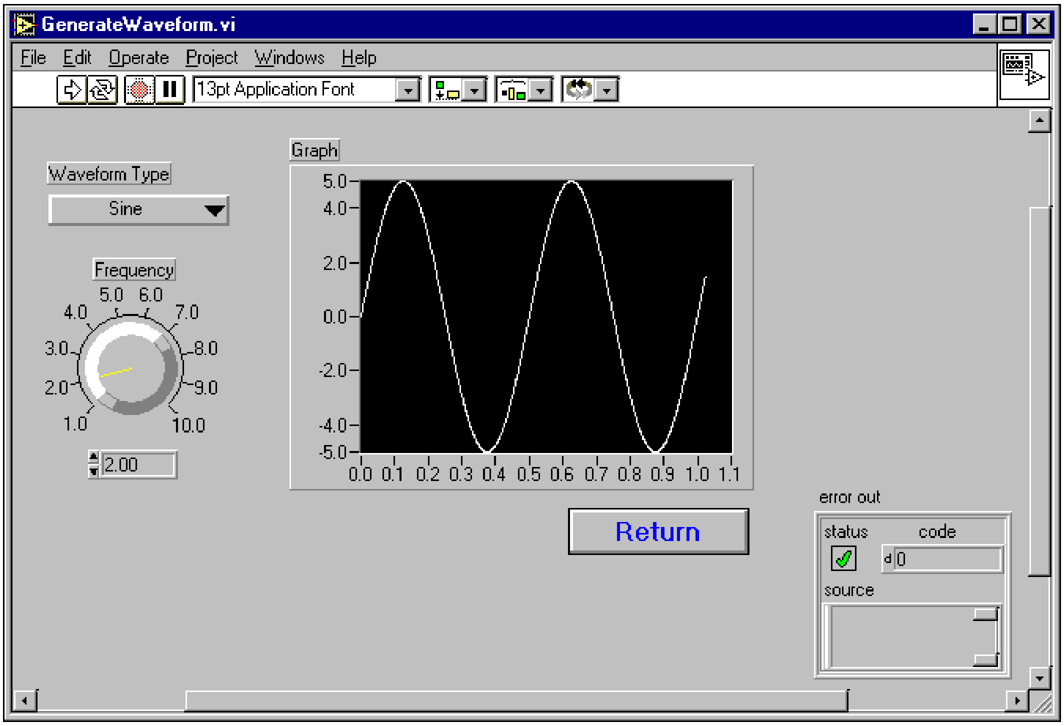


Figure 10-3. GenerateWaveform.vi Front Panel

16. Save the VI file as `GenerateTestStandWaveform.vi` in the `TestStand\Tutorial` directory by selecting the **File»Save As** in LabVIEW.

    You are saving the file in this manner to preserve the original VI for the next person that uses this tutorial. Later, you change the specified module in the TestStand step to point to this new VI.

    When you install TestStand, the installation adds a `TestStand.LLB` to the `LabVIEW\User.lib` directory. This VI library contains the TestStand LabVIEW Standard Prototype controls, and additional VIs to assist you when calling the TestStand ActiveX automation server.

Figure 10-4 shows the TestStand control palette.

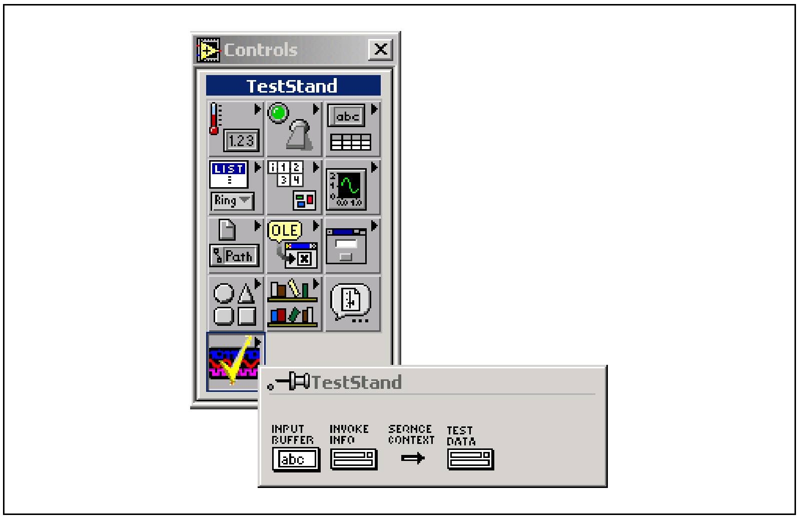


Figure 10-4. TestStand Control Palette

All test VIs that the LabVIEW Standard Prototype Adapter calls must contain Test Data and error out controls. Test VIs also might contain the Sequence Context, Input Buffer, and Invocation Info controls.

17. Add additional controls to the front panel so that the LabVIEW Standard Prototype Adapter can call the VI from TestStand, as follows:
    a. Drag the Sequence Context and Test Data controls from the TestStand control palette to the VI front panel.

b. Wire the two new controls to the terminal connector in the upper right corner of the front panel window, as shown in Figure 10-5. Wire the Sequence Context to the upper left position. Wire the Test Data cluster to the upper right position. The position of the connections is not rigidly enforced; however, by convention, you should wire controls to the left side of the connector pane and indicators to the right side of the connector pane. If you do not wire the controls to the terminal, TestStand cannot pass data to the controls.

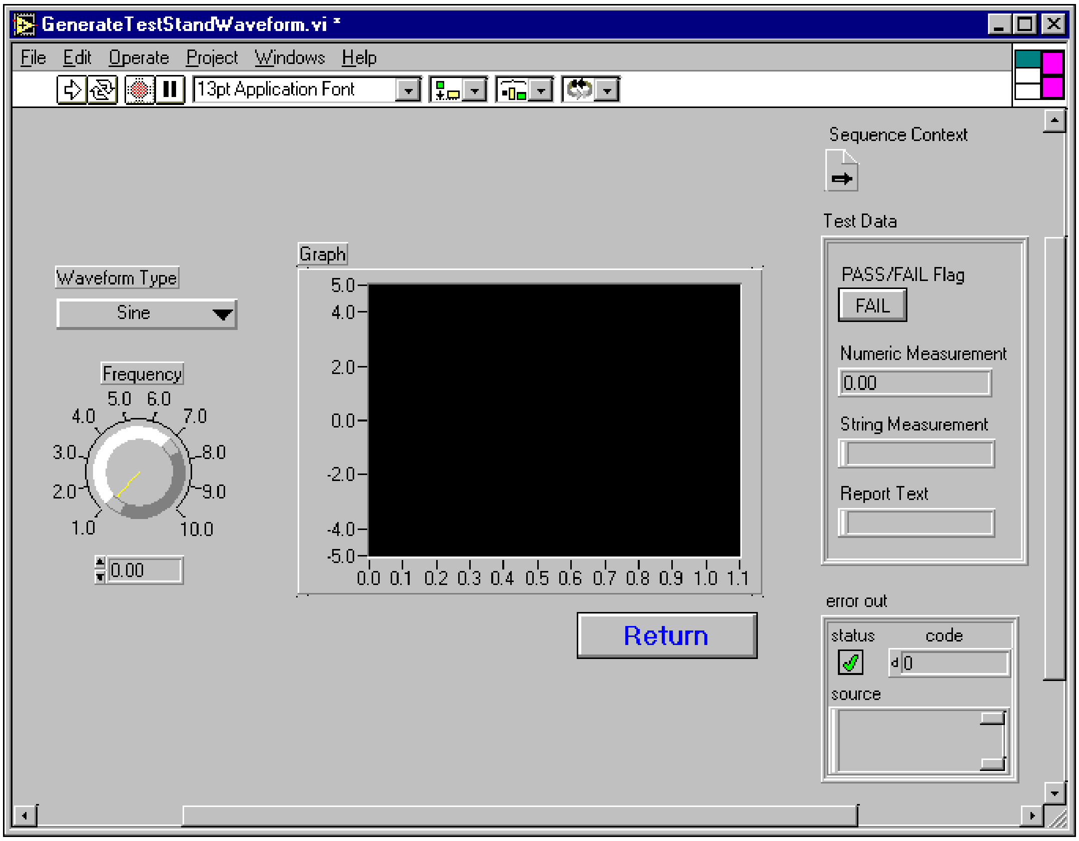


Figure 10-5. GenerateTestStandWaveform.vi Control Panel

c. The LabVIEW Standard Prototype Adapter does not allow you to pass array data to the Test VI directly through a VI terminal. Instead, the test VI must give the TestStand engine the array data using the TestStand API.

18. Use functions in the TestStand function palette to assign the array data to a TestStand variable using the sequence context refnum, as follows:

    a. Open the block diagram for the test VI as shown in Figure 10-6.

Note The three VIs, `InitializeTerminationMonitor.vi`, `GetMonitorStatus.vi`, and `CloseTerminationMonitor.vi`, monitor the status of the TestStand execution under which the step module is called. When performing a task inside of a step, such as displaying a dialog box, you should monitor the state of the current TestStand execution. If the execution is terminating or aborting, you should abort the task you are performing within the step module.

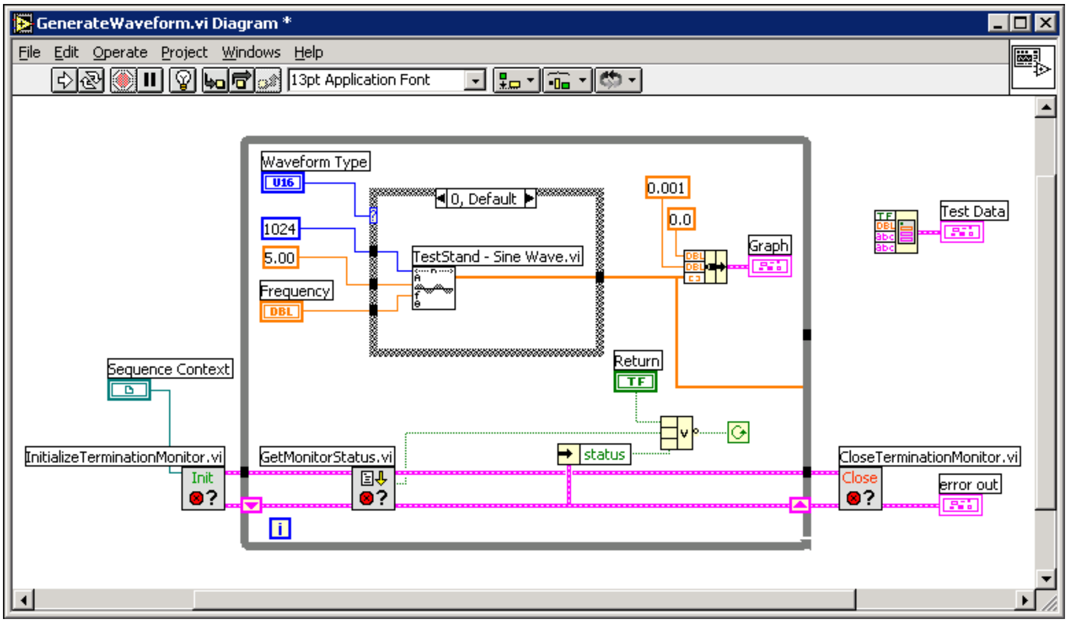


Figure 10-6. GenerateWaveform.vi Block Diagram

You can access the TestStand function palette from the user libraries in LabVIEW as shown in Figure 10-7.

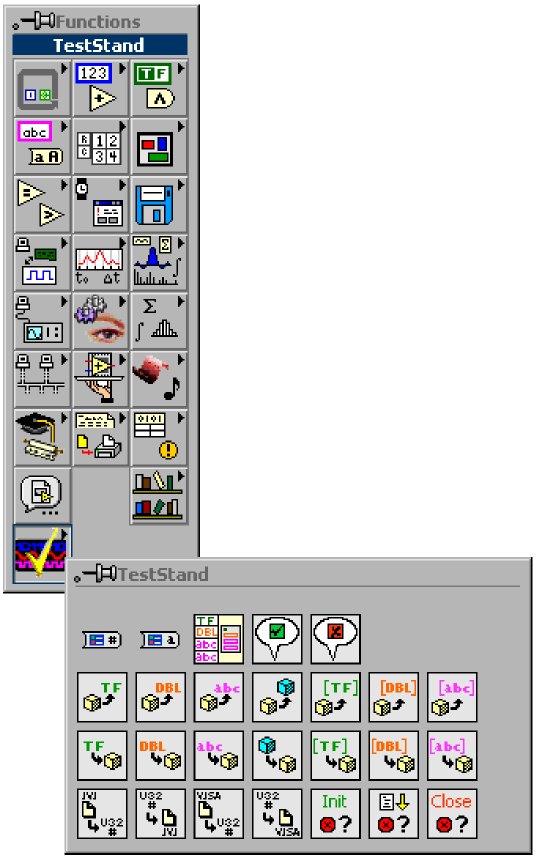


Figure 10-7. TestStand Function Palette

b. Using the TestStand VIs `Set Property Value (Numeric Array).vi` and `Create Test Data Cluster.vi`, update the block diagram as shown in Figure 10-8.

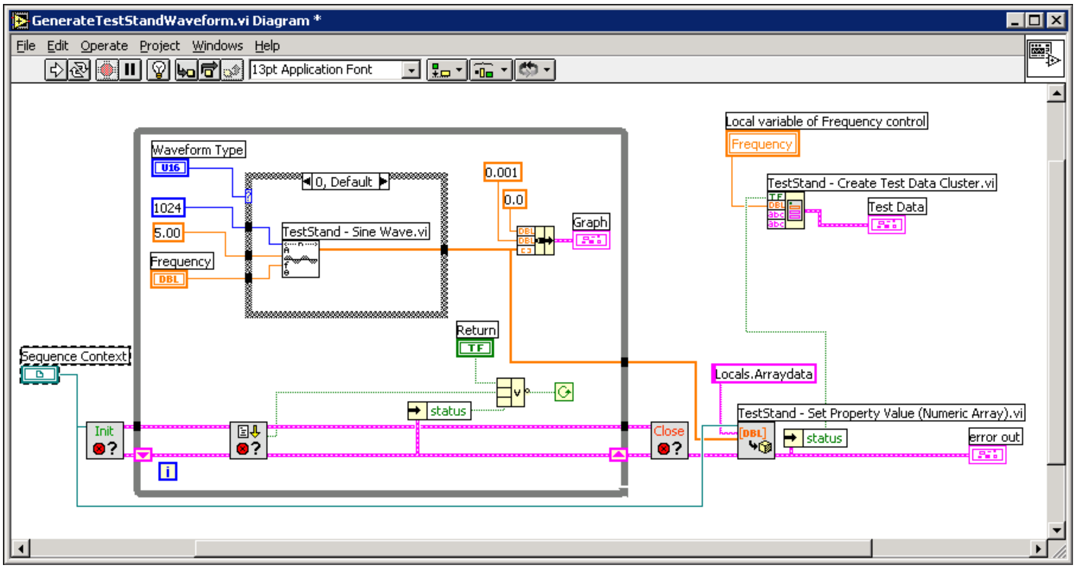

GenerateTestStandWaveform.vi Diagram *
File Edit Operate Project Windows Help
13pt Application Font
Waveform Type
U16
1024
5.00
Frequency
DBL
0, Default
TestStand - Sine Wave.vi
0.001
0.0
Graph
Return
TF
status
Sequence Context
Init
Close
Local variable of Frequency control
Frequency
TestStand - Create Test Data Cluster.vi
Test Data
Locals.Arraydata
TestStand - Set Property Value (Numeric Array).vi
status
error out

Figure 10-8. GenerateTestStandWaveform.vi Block Diagram

Note The TestStand function palette contains only wrapper VIs for the most commonly used TestStand ActiveX API methods and properties. Refer to the *TestStand Programmer Help* for more details on the entire ActiveX API.

c. Save your changes to the test VI by selecting **File»Save** in LabVIEW. Leave the test VI open in LabVIEW.

19. Return to the Edit LabVIEW VI Call dialog box in the sequence editor.
20. Re-specify the VI module for the step by clicking on the **Browse** button and selecting the file `GenerateTestStandWaveform.vi` in the `TestStand\Tutorial` directory.
21. Click on **OK** to close the Edit LabVIEW VI Call dialog box.
22. Create a second step that plots the data stored in the TestStand variable, as follows:

    a. Right click in the step list below the Get Array Data From LabVIEW step and select **Insert Step»Action** to insert a new Action step.

    b. Rename the new step `Display Array Data in LabVIEW`.

    c. Right click on the new step and select the **Specify Module** command in the context menu to display the Edit LabVIEW VI Call dialog box.

d. Enter the name `DisplayWaveform.vi` for the VI Pathname control.

e. Enable the options Sequence Context ActiveX Pointer and Show VI Front Panel When Called.

f. Click on the **Create Code** button to create and open the test VI in LabVIEW. The sequence editor prompts you to choose the directory in which to create the new VI.

g. Select the `TestStand\Tutorial` directory. Notice that the new VI already contains the Sequence Context, Test Data, and error out controls.

h. Add a Waveform Graph control and a Dialog button to the VI front panel, as shown in Figure 10-9.

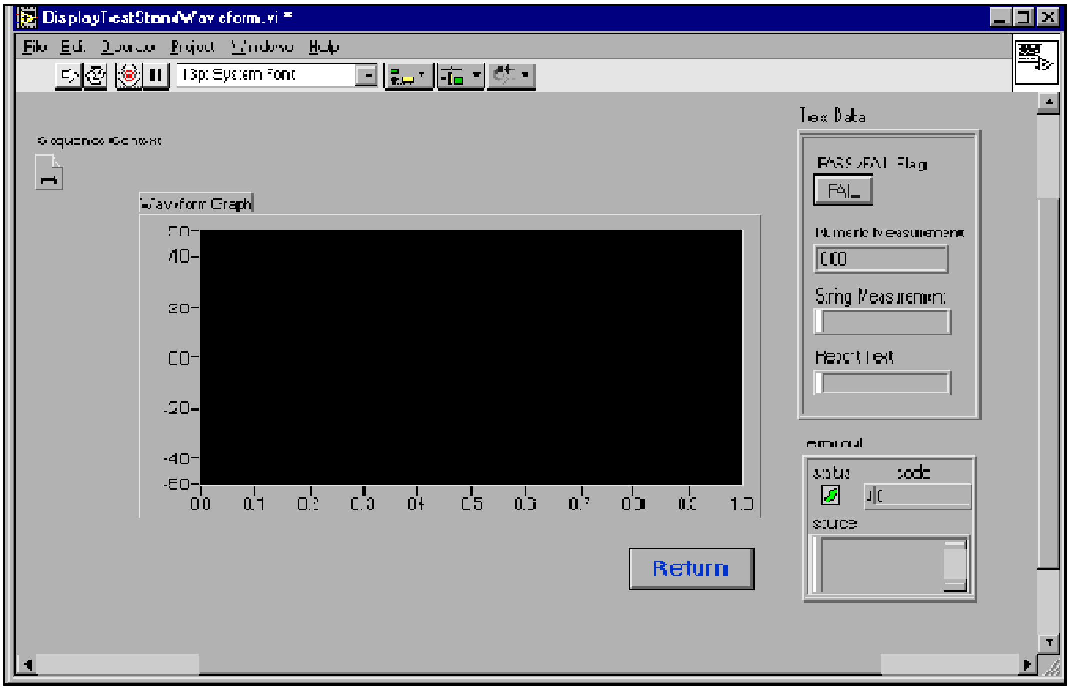


Figure 10-9. DisplayTestStandWaveform.vi Front Panel

i. Using the TestStand VIs `Get Property Value (Numeric Array).vi`, `Create Test Data Cluster.vi`, `InitializeTerminationMonitor.vi`, `GetMonitorStatus.vi`, and `CloseTermination Monitor.vi`, update the block diagram as shown in Figure 10-10.

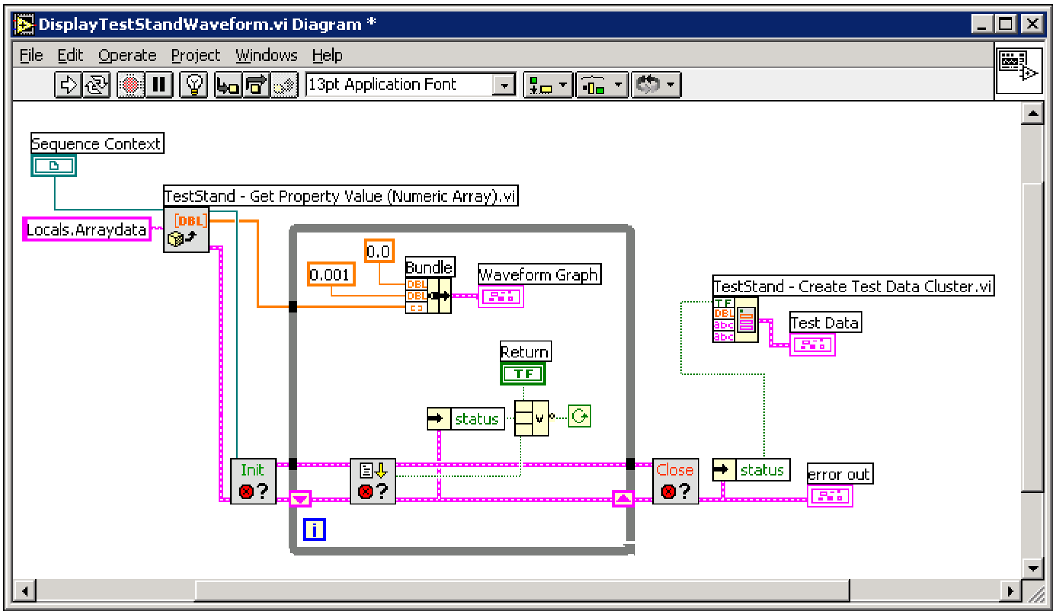

DisplayTestStandWaveform.vi Diagram *
File Edit Operate Project Windows Help
13pt Application Font
Sequence Context
TestStand - Get Property Value (Numeric Array).vi
Locals.Arraydata
0.0
0.001
Bundle
Waveform Graph
TestStand - Create Test Data Cluster.vi
Test Data
Return
status
Init
Close
status
error out

Figure 10-10. DisplayTestStandWaveform.vi Block Diagram

j. Save your changes to the test VI by selecting **File»Save** in LabVIEW. Leave the test VI open in LabVIEW.

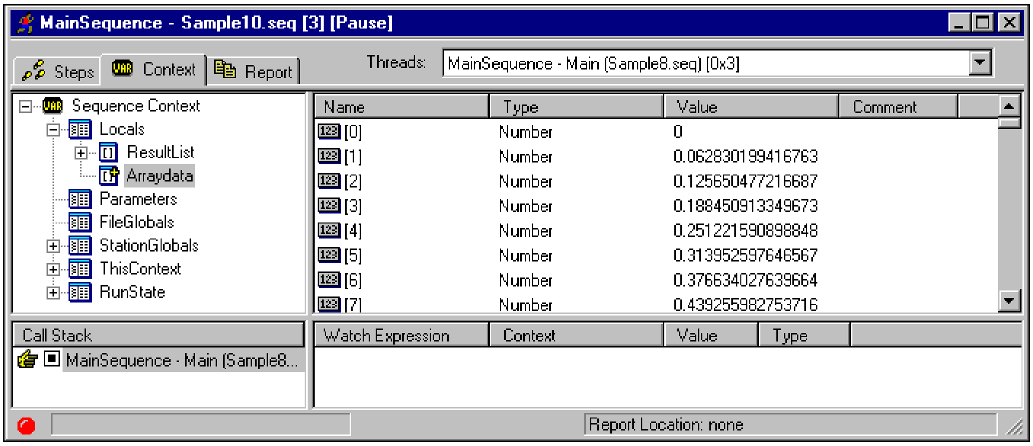
23. Return to the Edit LabVIEW VI Call dialog box in the sequence editor.
24. Click on **OK** to close the dialog box.
25. From the main sequence editor window, select **File»Save** to save the sequence file changes.

Running the Sequence

Run the sequence to verify that the code module properly generates and displays the array data.

1. Execute the sequence by selecting **Execute»Run MainSequence**.

2. When the front panel for the two VI tests appears, click on the **Return** button to continue. Notice that the second VI front panel displays the waveform you specified on the first front panel.
3. Close the Execution window after the execution completes.
4. Rerun the sequence and view the data stored in the TestStand array variable as follows:
   a. Place a breakpoint on the Display Array Data in LabVIEW step by right clicking on the step in the Sequence File window and selecting **Toggle Breakpoint** from the context menu.
   b. Execute the sequence again by selecting **Execute»Run MainSequence**.
   c. When the front panel for the first VI test appears, click on the **Return** button to continue.
   d. When the execution stops at the breakpoint, click on the Context tab and select the `Locals.Arraydata` variable, as shown in Figure 10-11. Notice the non-zero values for the contents of the array.


MainSequence - Sample10.seq [3] [Pause]

Steps | Context | Report

Threads: MainSequence - Main (Sample8.seq) [0x3]

- Sequence Context
  - Locals
    - ResultList
    - Arraydata
  - Parameters
  - FileGlobals
  - StationGlobals
  - ThisContext
  - RunState

Name	Type	Value	Comment
[0]	Number	0	
[1]	Number	0.062830199416763	
[2]	Number	0.125650477216687	
[3]	Number	0.188450913349673	
[4]	Number	0.251221590898848	
[5]	Number	0.313952597646567	
[6]	Number	0.376634027639664	
[7]	Number	0.439255982753716	

Call Stack

MainSequence - Main (Sample8...

Watch Expression	Context	Value	Type

Report Location: none

Figure 10-11. Locals.Arraydata

5. Select **Debug»Resume** to continue.
6. When the front panel for the second VI test appears, click on the **Return** button to finish the execution.
7. Close the Execution window after the execution completes.
8. Close all windows in the sequence editor.

If you develop code modules in LabWindows/CVI, proceed to the next section in this chapter and complete the *Setting Up the Example* section. Otherwise, go directly to Chapter 11, *Additional Development Features*, in this manual.

Using ActiveX in LabWindows/CVI Code Modules

In this exercise, you create a new sequence with two LabWindows/CVI steps that generate an array of data, and plot the data. This session of the tutorial assumes a general familiarity with the LabWindows/CVI development environment.

Note Verify that you are using the appropriate version of LabWindows/CVI with TestStand. Refer to the `readme.txt` file in the `TestStand\Doc` directory for more details.

Setting Up the Example

Close all windows in the sequence editor so you can complete this tutorial session.

Creating the Sequence and Tests

In this exercise, you create a new sequence in the sequence editor.

1. Open a new sequence by selecting **File»New Sequence File**.
2. Save the sequence by selecting **File»Save As**. Save the sequence as `Sample11.seq` in the `TestStand\Tutorial` directory. By saving the sequence file now, you can enter relative paths to code modules instead of absolute paths.
3. Click on the Locals tab of the Sequence File window.

4. Right click in the right pane and insert a numeric array variable, as shown in Figure 10-12.

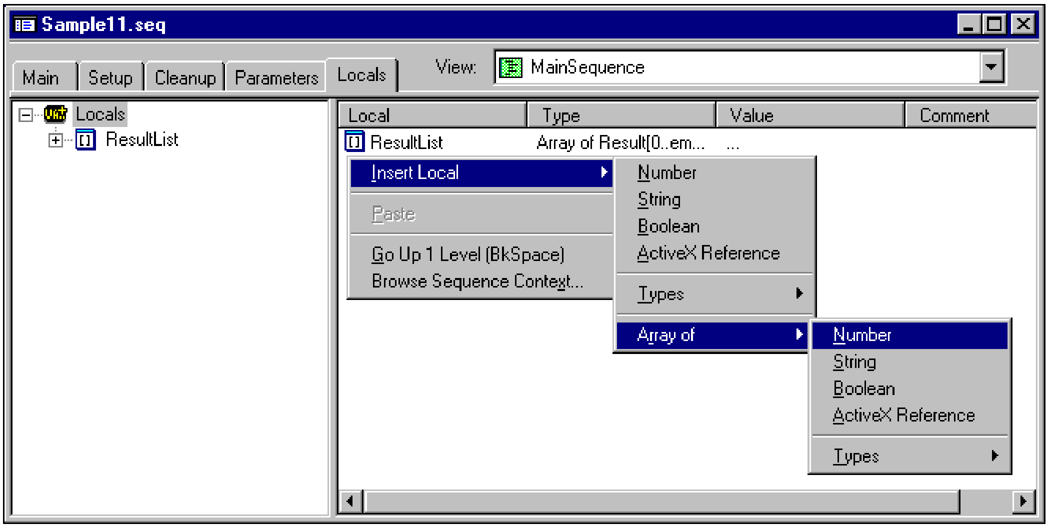
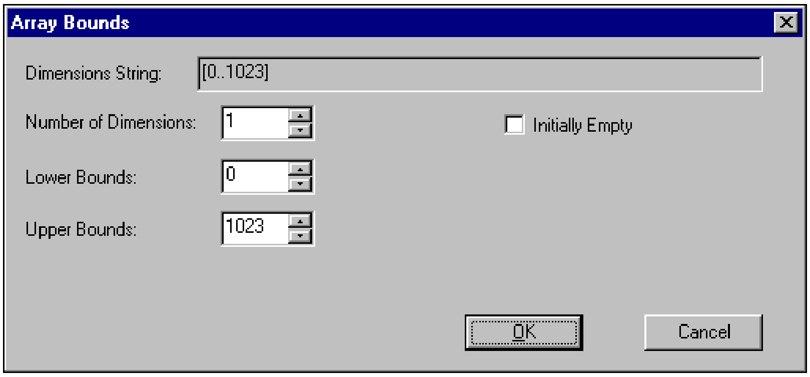


Figure 10-12. Insert Locals Array of Numeric

When you make this selection, the sequence editor displays the Array Bounds dialog box.

5. Enter a value of `1023` for the Upper Bounds control, as shown in Figure 10-13.


Array Bounds

Dimensions String: [0..1023]

Number of Dimensions: 1

Initially Empty ☐

Lower Bounds: 0

Upper Bounds: 1023

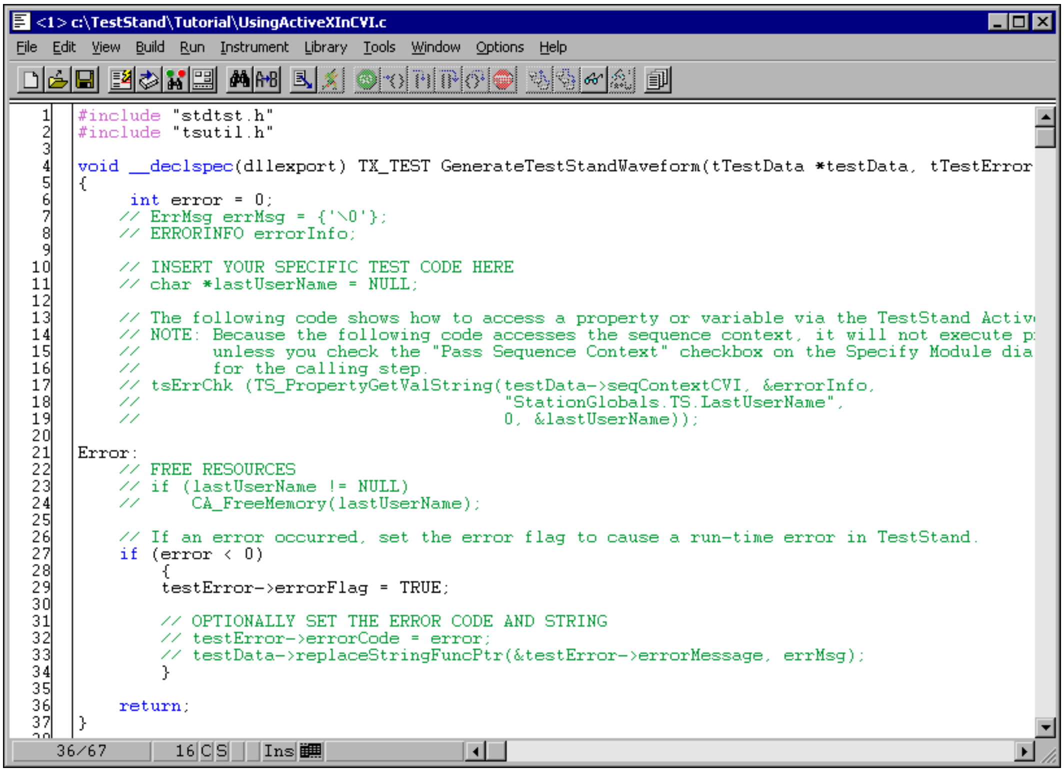
Figure 10-13. Array Bounds Dialog Box

6. Click on **OK** to close the dialog box.
7. Right click on the variable and select **Rename** from the context menu.
8. Rename the variable `Arraydata`.
9. Click on the Main tab of the Sequence File window. Select C/CVI Standard Prototype Adapter in the Adapter Selector Ring control.
10. Right click in the step list and select **Insert Step»Action** to insert a new Action step.
11. Rename the step `Get Array Data From CVI`.
12. Right click on the new step and select the **Specify Module** command in the context menu.
13. On the Module tab in the Module Type ring control, select Dynamic Link Library (`*.dll`).
14. For the Module Pathname control, enter the name `UsingActiveXInCVI.DLL`.
15. For the Function Name control, enter the name `GenerateTestStandWaveform`.
16. Enable the Pass Sequence Context option.
17. On the Source Code tab, click on the **Create Code** button. When the adapter prompts you for a pathname for the project, enter `UsingActiveXInCVI.prj` in the `TestStand\Tutorial` directory.
18. When the dialog box prompts you for a pathname for the source file, enter the name `UsingActiveXInCVI.c`.

    After you enter the name, TestStand does the following:

    1. Launches an external instance of LabWindows/CVI
    2. Creates a new project file in LabWindows/CVI
    3. Creates the source file
    4. Adds the source file and the TestStand support instrument drivers to the project
    5. Generates a template `GenerateTestStandWaveform` function in the source file

Figure 10-14 shows the generated function in the source file.



<1> c:\TestStand\Tutorial\UsingActiveXInCVI.c
File Edit View Build Run Instrument Library Tools Window Options Help
36/67
16 C S
Ins

```
#include "stdtst.h"
#include "tsutil.h"

void __declspec(dllexport) TX_TEST GenerateTestStandWaveform(tTestData *testData, tTestError
{
     int error = 0;
    // ErrMsg errMsg = {'\0'};
    // ERRORINFO errorInfo;

    // INSERT YOUR SPECIFIC TEST CODE HERE
    // char *lastUserName = NULL;

    // The following code shows how to access a property or variable via the TestStand Activ
    // NOTE: Because the following code accesses the sequence context, it will not execute p
    //       unless you check the "Pass Sequence Context" checkbox on the Specify Module dia
    //       for the calling step.
    // tsErrChk (TS_PropertyGetValString(testData->seqContextCVI, &errorInfo,
    //                                    "StationGlobals.TS.LastUserName",
    //                                    0, &lastUserName));

Error:
    // FREE RESOURCES
    // if (lastUserName != NULL)
    //     CA_FreeMemory(lastUserName);

    // If an error occurred, set the error flag to cause a run-time error in TestStand.
    if (error < 0)
        {
        testError->errorFlag = TRUE;

        // OPTIONALLY SET THE ERROR CODE AND STRING
        // testError->errorCode = error;
        // testData->replaceStringFuncPtr(&testError->errorMessage, errMsg);
        }

    return;
}
```

Figure 10-14. Generated GenerateTestStandWaveform Source

Note If someone else previously completed this session of the tutorial, the project and source files might already exist. Replace any existing copies of the files. When LabWindows/CVI prompts you to replace the existing function, click on the **Replace** button to continue with this tutorial session.

19. Update the `GenerateTestStandWaveform` function to prompt an operator to enter the number of sine cycles to initialize the array with. Update the source code for the function as follows. Changed lines appear in bold.

```
void __declspec(dllexport) TX_TEST
  GenerateTestStandWaveform(tTestData *testData,
  tTestError *testError)
{
    int error = 0;
    ErrMsg errMsg = {"Error occurred generating
      waveform"};
    ERRORINFO errorInfo;

    int i;
    double cycles = 2.0;
    char buffer[32];
    double Arraydata[1024];
    VARIANT variantData;

    // Prompt for number of cycles
    PromptPopup ("Frequency", "Please input number of
      cycles for the array?", buffer, 32);
    sscanf(buffer, "%lf", &cycles);

    // Initialize C array
    for (i=0; i<1024; i++)
        Arraydata[i] = sin((2*3.14) * i * cycles /
        1024);

    // Copy C array to VARIANT
    CA_VariantSet1DArray (&variantData, CAVT_DOUBLE,
      1024, Arraydata);

    // The following code shows how to access a
    // property or variable via the TestStand
    // ActiveX API
    tsErrChk (TS_PropertySetValVariant
      (testData->seqContextCVI, &errorInfo,
      "Locals.Arraydata", 0, variantData));

Error:
    // FREE RESOURCES

    // If an error occurred, set the error flag to
    //   cause a run-time error in TestStand.
    if (error < 0)
    {
        testError->errorFlag = TRUE;

        // OPTIONALLY SET THE ERROR CODE AND STRING
        testError->errorCode = error;
        testData->replaceStringFuncPtr(&testError->
        errorMessage, errMsg);
    }

    return;
}
```

20. Compile the source code by selecting **Build»Compile File** to verify that your changes are correct.
21. Save the source code after you successfully compile.
22. Add another function that gets the array data from the TestStand engine and plots the data. Type the following additional function into the source file.

```
void __declspec(dllexport) TX_TEST
  DisplayTestStandWaveform(CAObjHandle
  seqContextCVI)
{
    int error = 0;
    ErrMsg errMsg = {'\0'};
    ERRORINFO errorInfo;

    int elements = 0;
    double *Arraydata = NULL;
    VARIANT variantData;

    // The following code shows how to access a
    // property or variable via the TestStand
    // ActiveX API
    tsErrChk (TS_PropertyGetValVariant
      (seqContextCVI, &errorInfo, "Locals.Arraydata",
      0, &variantData));

    // Copy C array to VARIANT
    CA_VariantGet1DArray (&variantData, CAVT_DOUBLE,
      &Arraydata, &elements);

    WaveformGraphPopup ("Waveform From TestStand",
      Arraydata, elements, VAL_DOUBLE, 1.0, 0.0,
      0.0, 1.0);

Error:
    if (Arraydata)
          CA_FreeMemory(Arraydata);

    return;
}
```

23. Compile the source code by selecting **Build»Compile File** to verify that your changes are correct.
24. Save the source code after you successfully compile.
25. To build the DLL, select **Build»Create Debuggable Dynamic Link Library** in the Project window.
26. After the DLL build is complete, return to the sequence editor.

27. Click on **OK** to close the Edit C/CVI Module Call dialog box to return to the Main step group view.

Note If LabWindows/CVI returns a file permission error when creating the DLL, return to the sequence editor and select the Unload All Modules command from the **File** menu. When you make this selection, TestStand unloads all step code modules, which includes DLLs, VIs and any other modules the adapter loads. Return to LabWindows/CVI and rebuild the DLL.

28. Select the DLL Flexible Prototype Adapter in the Adapter Selector Ring control.
29. Right click in the step list below the Get Array Data From CVI step and select **Insert Step»Action** to insert a new Action step.
30. Rename the step `Display Data In CVI`.
31. Right click on the new step and select the **Specify Module** command in the context menu.
32. On the Module tab, click on the **Browse** button next to the DLL Pathname control.
33. Select the `UsingActiveXInCVI.dll` that you created.
34. Select the `DisplayTestStandWaveform` function in the Function Name ring control. TestStand displays a message saying the function does not have parameter information in the DLL.
35. Click on the **New** button to create a parameter for the function.
36. Rename the `arg1` parameter `seqContextCVI`.
37. Select Object in the Category ring control.
38. Select CVI ActiveX Automation Handle in the Object Type ring control.
39. Enter the expression, `ThisContext`, in the Value Expression control.

Note The purpose of this session is to demonstrate how to use the sequence context within a code module. When you use the DLL Flexible Prototype Adapter, you can pass the array directly to the DLL as a parameter to the function.

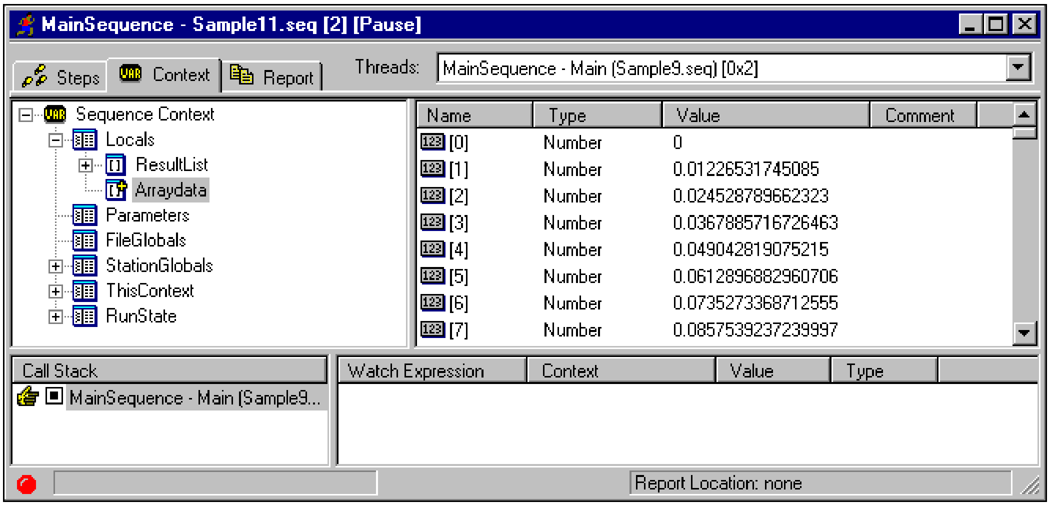
40. Click on **OK** to close the Edit DLL Call dialog box.
41. In the main sequence editor window, select **File»Save** to save the sequence file changes.

Running the Sequence

In this exercise, you run the sequence to verify that the code module properly generates and displays the array data.

1. Execute the sequence by selecting **Execute»Run MainSequence**.
2. Enter a value of `5` for the number of sine cycles to generate for the array data.
3. Click on the **OK** button to continue.
4. After the second step displays the graph, click on **OK** to continue.
5. Close the Execution window after the execution completes.
6. Rerun the sequence execution and view the data stored in the array variable in TestStand as follows:
    a. Place a breakpoint on the `Display Data In CVI` step by right clicking on the step in the Sequence File window and selecting **Toggle Breakpoint** from the context menu.
    b. Execute the sequence again by selecting **Execute»Run MainSequence**.
    c. Enter a value of `5` for the number cycles to generate in the array.
    d. Click on the **OK** button to continue.
    e. When the execution stops at the breakpoint, click on the Context tab.

f. Select the `Locals.Arraydata` variable as shown in Figure 10-15. Notice the non-zero values for the contents of the array.



Name	Type	Value	Comment
[0]	Number	0	
[1]	Number	0.01226531745085	
[2]	Number	0.024528789662323	
[3]	Number	0.0367885716726463	
[4]	Number	0.049042819075215	
[5]	Number	0.0612896882960706	
[6]	Number	0.0735273368712555	
[7]	Number	0.0857539237239997	

Figure 10-15. Locals.Arraydata Values

7. Select **Debug»Resume**.
8. After the graph window appears, click on **OK** to continue.
9. Close the Execution window after the execution completes.
10. Close all windows in the sequence editor.

This concludes this tutorial session. In the next session, you learn more advanced features of TestStand to use when developing and debugging sequences.

11 Additional Development Features

In this chapter, you learn how to use more advanced features available when developing and debugging sequences. Also, you learn how to interactively execute steps, and how to dynamically call a sequence by name.

Setting Up the Example

Close all windows in the sequence editor so you can complete this tutorial session.

Interactive Execution

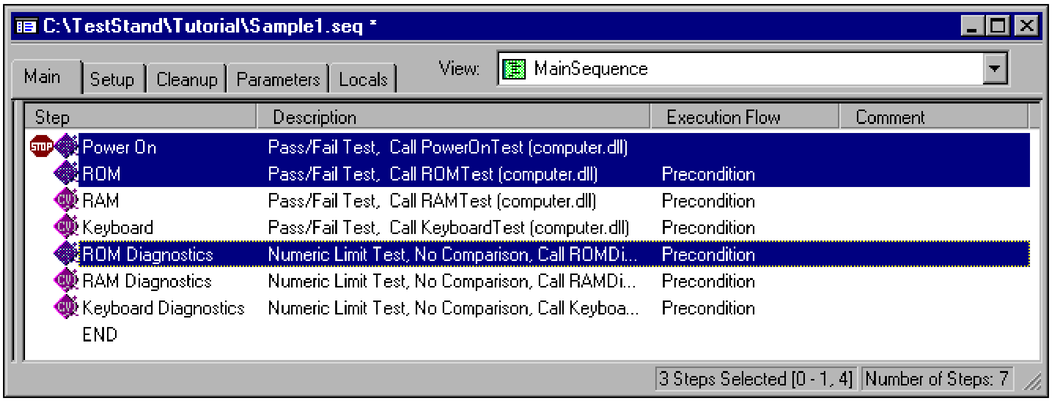
In this exercise, you run selected steps from a Sequence File window and interactively execute steps while paused at a breakpoint during an execution.

Running Selected Steps as a Separate Execution

In this exercise, you execute selected steps in a Sequence File window.

1. Open `Sample1.seq` from the `TestStand\Tutorial` directory.
2. After you open the sequence file, place a breakpoint on the Power On step by clicking to the left of the step icon or by right clicking on the step and selecting **Toggle Breakpoint** from the context menu.
3. Select the Power On, ROM, and ROM Diagnostics steps by holding down the <Ctrl> key and clicking on each step.

4. After you make these selections, the Sequence File window appears, as shown in Figure 11-1.



Step	Description	Execution Flow	Comment
Power On	Pass/Fail Test, Call PowerOnTest (computer.dll)		
ROM	Pass/Fail Test, Call ROMTest (computer.dll)	Precondition	
RAM	Pass/Fail Test, Call RAMTest (computer.dll)	Precondition	
Keyboard	Pass/Fail Test, Call KeyboardTest (computer.dll)	Precondition	
ROM Diagnostics	Numeric Limit Test, No Comparison, Call ROMDi...	Precondition	
RAM Diagnostics	Numeric Limit Test, No Comparison, Call RAMDi...	Precondition	
Keyboard Diagnostics	Numeric Limit Test, No Comparison, Call Keyboa...	Precondition	
END			

3 Steps Selected [0 - 1, 4] Number of Steps: 7

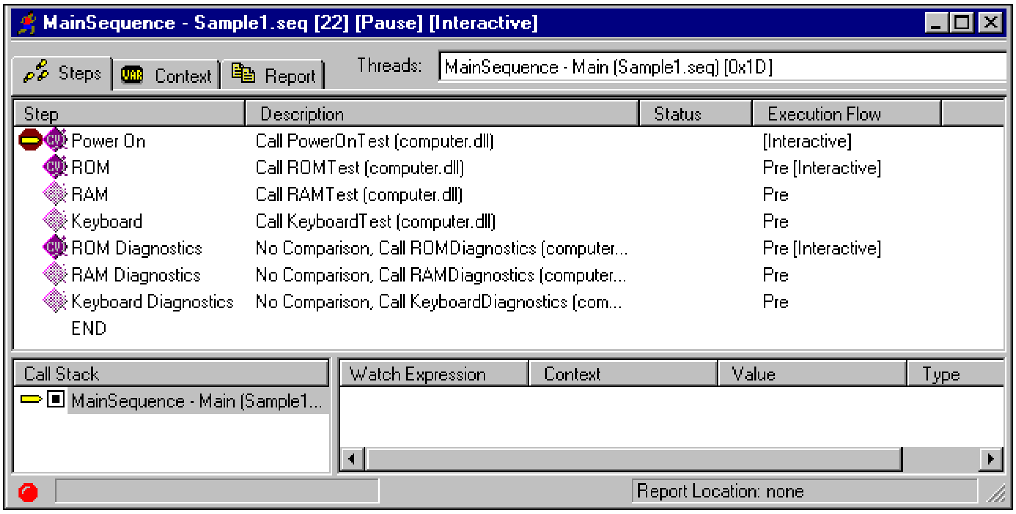
Figure 11-1. Selecting Multiple Steps in a Sequence File Window

5. Select **Execute»Run Selected Steps**. When you make this selection, TestStand starts a new execution.

   When you run selected steps from a Sequence File window, by default TestStand executes the Setup and Cleanup step groups and the steps in the Main step group.

6. When the Test Simulator dialog box appears, click on **Done** to close the dialog box.

7. The execution now enters the breakpoint on the Power On step, as shown in Figure 11-2.



MainSequence - Sample1.seq [22] [Pause] [Interactive]

Threads: MainSequence - Main (Sample1.seq) [0x1D]

Step	Description	Status	Execution Flow
Power On	Call PowerOnTest (computer.dll)		[Interactive]
ROM	Call ROMTest (computer.dll)		Pre [Interactive]
RAM	Call RAMTest (computer.dll)		Pre
Keyboard	Call KeyboardTest (computer.dll)		Pre
ROM Diagnostics	No Comparison, Call ROMDiagnostics (computer...		Pre [Interactive]
RAM Diagnostics	No Comparison, Call RAMDiagnostics (computer...		Pre
Keyboard Diagnostics	No Comparison, Call KeyboardDiagnostics (com...		Pre
END			

Call Stack
MainSequence - Main (Sample1...

Watch Expression	Context	Value	Type

Report Location: none

Figure 11-2. Breakpoint During Interactive Execution

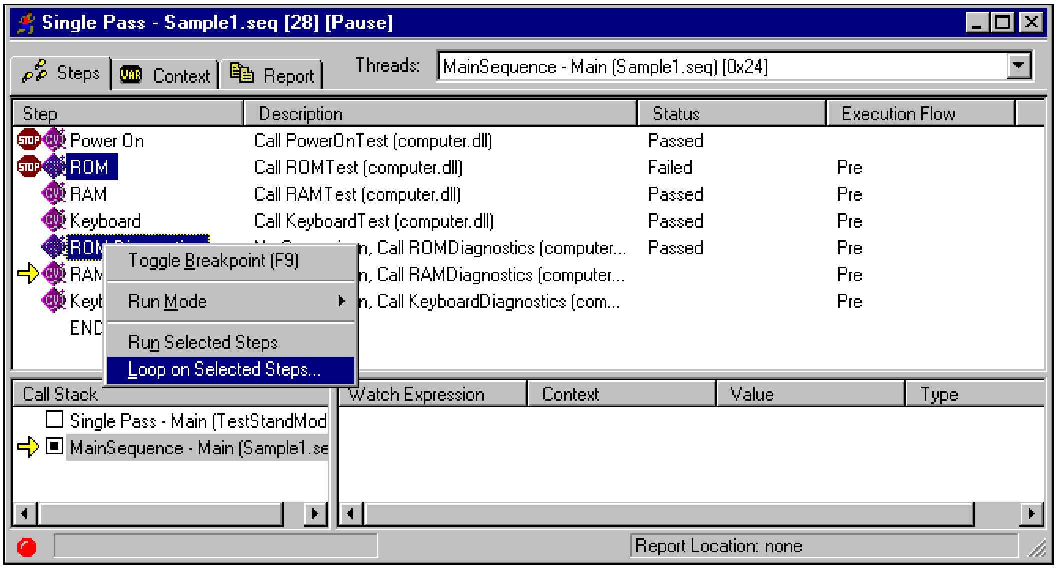
Notice that the pointer for the interactive execution is a narrow arrow instead of the yellow arrow that the sequence editor uses for a normal execution.

8. Single-step the execution twice by selecting **Debug»Step Over**. Notice that the execution executes only the steps that you previously selected and that the non-selected step icons are dimmed.
9. Complete the execution by selecting **Debug»Resume**. Notice that TestStand ignores the preconditions for the ROM Diagnostics step and runs the step even though the ROM step passed.
10. Close the Execution window after the execution completes.
11. Repeat steps 3 through 10 but this time select **Execute»Run Selected Steps Using»Single Pass** in step 5. TestStand executes your steps using the process model entry point Single Pass which produces a UUT report.

Running Selected Steps During an Execution

In this exercise, you interactively execute selected steps while paused at a breakpoint during an execution.

1. Select **Execute»Single Pass** to start a new execution.
2. When the Test Simulator dialog box appears, select the ROM test to fail.
3. Click on **Done** to close the dialog box. The execution now enters the breakpoint on the Power On step.
4. Using **Debug»Step Over**, continue the execution until you reach the RAM Diagnostics step. Notice that the ROM step failed.
5. Place a second breakpoint on the ROM step in the Execution window by clicking to the left of the step icon or by right clicking on the step and selecting **Toggle Breakpoint** from the context menu.
6. Now select the ROM, and ROM Diagnostics steps by holding down the <Ctrl> key and clicking on each step.
7. Right click on the ROM Diagnostics step and select **Loop on Selected Steps** from the context menu, as shown in Figure 11-3.



Single Pass - Sample1.seq [28] [Pause]

Threads: MainSequence - Main (Sample1.seq) [0x24]

Step	Description	Status	Execution Flow
Power On	Call PowerOnTest (computer.dll)	Passed	
ROM	Call ROMTest (computer.dll)	Failed	Pre
RAM	Call RAMTest (computer.dll)	Passed	Pre
Keyboard	Call KeyboardTest (computer.dll)	Passed	Pre
ROM	n, Call ROMDiagnostics (computer...	Passed	Pre
RAM	n, Call RAMDiagnostics (computer...		Pre
Keyb	n, Call KeyboardDiagnostics (com...		Pre
END			

Single Pass - Main (TestStandMod
MainSequence - Main (Sample1.se

Report Location: none

Figure 11-3. Loop on Selected Steps During Execution

8. In the Loop on Selected Steps dialog box, enter the number `100` in the Loop Count control.
9. Click on **OK** to close the dialog box. The sequence editor starts an interactive execution for the selected steps and enters a paused state on the breakpoint for the ROM step. Notice that the yellow arrow icon is still on the RAM Diagnostics step and a new narrow arrow icon is now pointing to the ROM step.
10. Single-step the interactive execution by selecting **Debug»Step Over**. Notice that the interactive execution toggles between only the ROM and the ROM Diagnostics steps.
11. The status of the ROM step does not pass and continues to fail. Rather than complete the 100 loops of the interactive execution, select **Debug»Terminate Interactive Execution**. When you make this selection, the TestStand returns the execution to a paused state on the RAM Diagnostics step.
12. Force the execution to continue from a step other than the currently paused step as follows:
    a. Click on the ROM step so that it is the only highlighted step.
    b. Right click on the ROM step and select **Set Next Step** from the context menu. Notice that the yellow arrow icon moves from the RAM Diagnostics step to the ROM step.
    c. Select **Debug»Step Over** to single-step once and notice that the execution executes the ROM step instead of the RAM Diagnostics step.
13. Complete the execution by selecting **Debug»Resume**.

    When the report file completes, notice that the report contains entries for each interactively executed step.
14. Close all windows in the sequence editor and do not save any changes to the sequence file.

Calling Sequences Dynamically and Passing Parameters

In the following exercise, you add a step to a sequence that dynamically runs one of two sequences.

Adding a Step to Sequence

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

1. Open `Sample1.seq` from the `TestStand\Tutorial` directory.
2. Click on the **Locals** tab of the sequence window.
3. Right click in the right pane and select **Insert Local»String** from the context menu.
4. Rename the local variable `CPUType`.
5. Click on the Main tab in the Sequence File window to display the steps in the Main step group.
6. Right click on the Power On step and select **Insert Step»Message Popup** from the context menu.
7. Rename the new step `Select CPU Type`.
8. Right click on the new step and select **Edit Message Settings** from the context menu.
9. Under the Text and Buttons tab, change the following control values on the Configure Message Box Step dialog box:

Title Expression	`"Select CPU"`
Message Expression	`"Please select the CPU Type for the UUT."`
Button 1	`"INTEL CPU"`
Button 2	`"AMD CPU"`
Cancel Button	`None`

10. Click on **OK** to close the Configure Message Box Step dialog box.
11. Select the Options tab and enable the Make Modal option. Enabling this option prevents the message popup dialog box from being hidden behind the Sequence Editor window and prevents the user from interacting with the Sequence Editor until the user closes the message popup dialog box.

12. Right click on the Select CPU Type step and select **Properties** from the context menu.
13. Click on the Expressions tab.
14. Enter the following expression in the Post Expression control. You can click the **Browse** button next to the Post Expression control and use the Expression Browser to create this expression. You can also get descriptions of condition operators such as ?:, in the Expression Browser.

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

    This expression assigns the string value `"AMD"` or `"INTEL"` to the local variable, depending on which button the user clicks on.
15. Click on **OK** to close the properties dialog box.
16. Right click on the Select CPU Type step and select the **Insert Step»Message Popup** command from the context menu.
17. Rename the new step `Specify Number of CPUs`.
18. Right click on the new step and select the **Edit Message Settings** command from the context menu.
19. Change the following control values on the Configure Message Box Step dialog box:

Title Expression	`"Number of CPUs"`
Message Expression	`"Please select the number of CPUs installed for the UUT."`
Button 1	`"1"`
Button 2	`"2"`
Button 3	`"3"`
Button 4	`"4"`
Cancel Button	`None`

20. Select the Options tab and enable the Make Modal option.
21. Click on **OK** to close the Configure Message Box Step dialog box.
22. Right click on the Specify Number of CPUs step and select the **Insert Step»Sequence Call** command in the context menu.
23. Rename the step `CPU Test`.
24. Right click on the new CPU Test step and select **Specify Module** from the context menu.

25. Enable the Specify Expressions for Pathname and Sequence option.
26. Enter the following values for the File Pathname Expression and Sequence Expression controls:

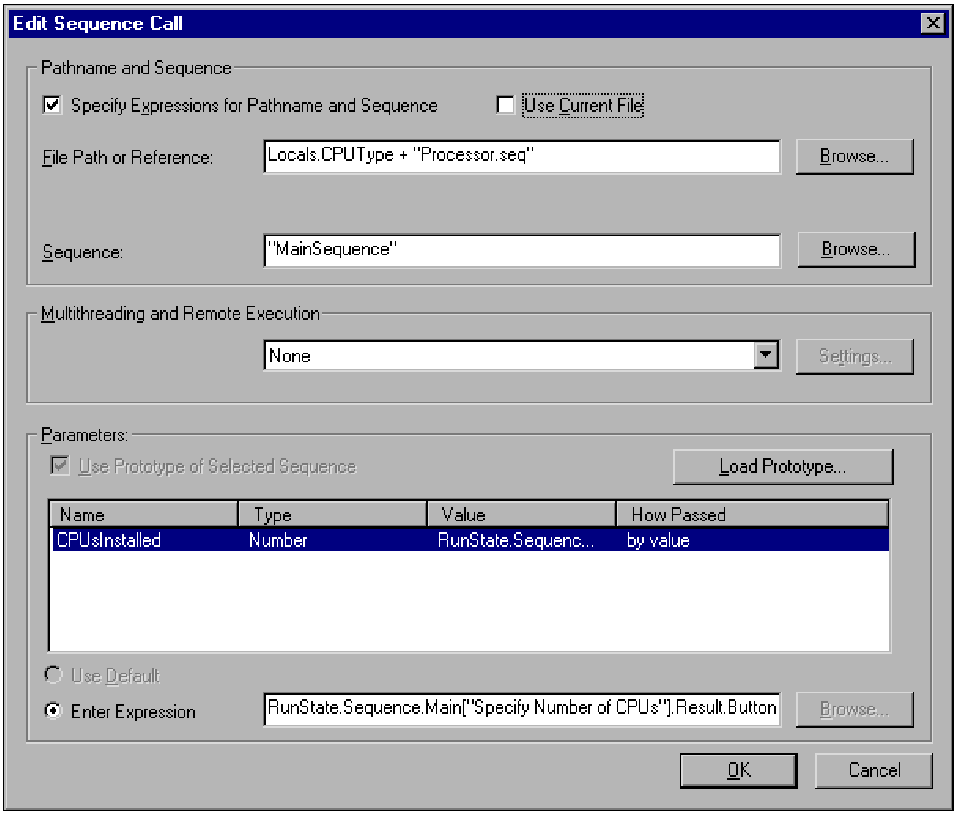
File Pathname Expression	`Locals.CPUType + "Processor.seq"`
Sequence Expression	`"MainSequence"`

27. Select the prototype for the sequence call by clicking on the **Load Prototype** button.
28. Click on the **Browse** button in the Load Sequence Prototype dialog box.
29. Select the `AMDProcessor.seq` sequence file.
30. Click on **OK** twice to close both the Select Sequence File and the Load Sequence Prototype dialog boxes.

    Notice that TestStand populates the Parameters section with the parameter list for the sequence.
31. Click on the `CPUsInstalled` parameter.
32. Select the Enter Expression option.
33. Enter the following expression into its string control, or click on **Browse** and find the property in the Expression Browser dialog box:

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

Figure 11-4 shows the completed Edit Sequence Call dialog box.



[✓] Specify Expressions for Pathname and Sequence

[ ] Use Current File

File Path or Reference: Locals.CPUType + "Processor.seq"

Sequence: "MainSequence"

Multithreading and Remote Execution: None

[✓] Use Prototype of Selected Sequence

Name	Type	Value	How Passed
CPUsInstalled	Number	RunState.Sequenc...	by value

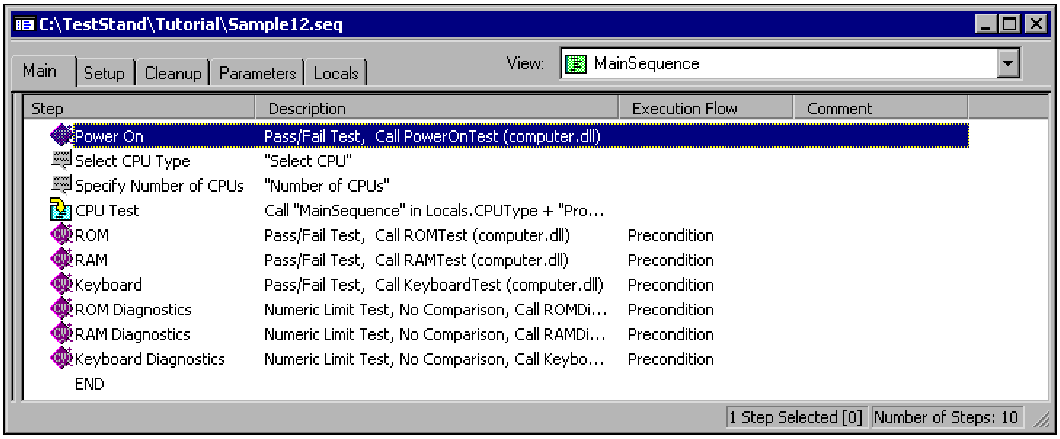
○ Use Default

● Enter Expression: RunState.Sequence.Main["Specify Number of CPUs"].Result.Button

Figure 11-4. Dynamically Calling with an Expression

34. Click on **OK** to close the dialog box.
35. Select **File»Save As** and save the sequence in the `TestStand\Tutorial` directory as `Sample12.seq`.

Figure 11-5 shows the resulting sequence.



Step	Description	Execution Flow	Comment
Power On	Pass/Fail Test, Call PowerOnTest (computer.dll)		
Select CPU Type	"Select CPU"		
Specify Number of CPUs	"Number of CPUs"		
CPU Test	Call "MainSequence" in Locals.CPUType + "Pro...		
ROM	Pass/Fail Test, Call ROMTest (computer.dll)	Precondition	
RAM	Pass/Fail Test, Call RAMTest (computer.dll)	Precondition	
Keyboard	Pass/Fail Test, Call KeyboardTest (computer.dll)	Precondition	
ROM Diagnostics	Numeric Limit Test, No Comparison, Call ROMDi...	Precondition	
RAM Diagnostics	Numeric Limit Test, No Comparison, Call RAMDi...	Precondition	
Keyboard Diagnostics	Numeric Limit Test, No Comparison, Call Keybo...	Precondition	
END			

1 Step Selected [0] Number of Steps: 10

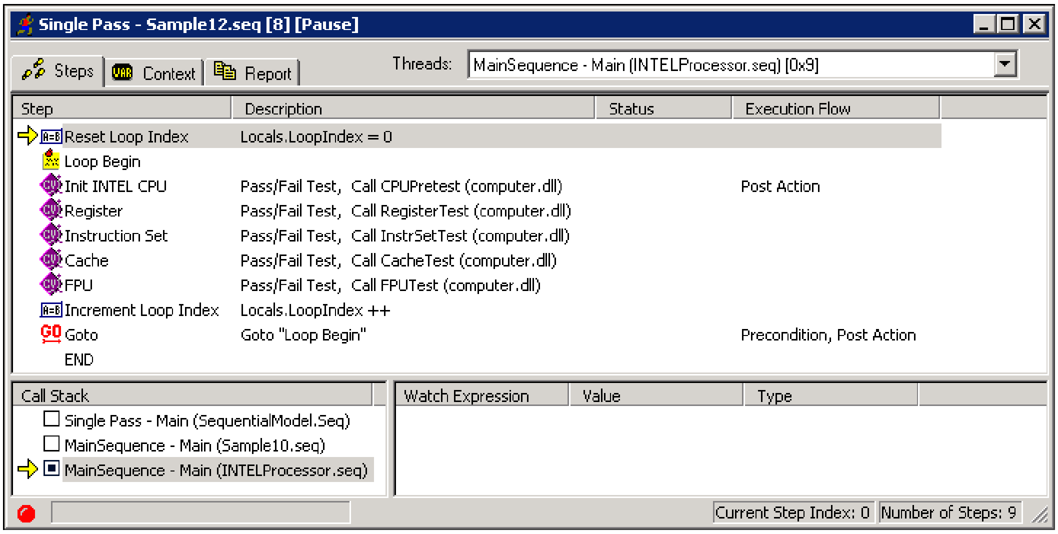
Figure 11-5. Dynamically Calling a Sequence

Running a Sequence

Complete the following steps to run a sequence dynamically:

1. Place a breakpoint on the CPU Test step by clicking to the left of the step icon or by right clicking on the step and selecting **Toggle Breakpoint** from the context menu.
2. Select **Execute»Single Pass**.
3. Click on **Done** in the Test Simulator prompt.
4. Click on the **INTEL CPU** button in the Select CPU prompt.
5. Click on the **2** button in the Number of CPUs prompt.

6. After the execution pauses at the breakpoint on the CPU Test step, single-step into the subsequence by selecting **Debug»Step Into**. Notice that the Call Stack pane lists `INTELProcessor.seq` at the bottom of the sequence call stack, as shown in Figure 11-6.


Single Pass - Sample12.seq [8] [Pause]

Steps | Context | Report — Threads: MainSequence - Main (INTELProcessor.seq) [0x9]

Step	Description	Status	Execution Flow
Reset Loop Index	Locals.LoopIndex = 0		
Loop Begin			
Init INTEL CPU	Pass/Fail Test, Call CPUPretest (computer.dll)		Post Action
Register	Pass/Fail Test, Call RegisterTest (computer.dll)		
Instruction Set	Pass/Fail Test, Call InstrSetTest (computer.dll)		
Cache	Pass/Fail Test, Call CacheTest (computer.dll)		
FPU	Pass/Fail Test, Call FPUTest (computer.dll)		
Increment Loop Index	Locals.LoopIndex ++		
Goto	Goto "Loop Begin"		Precondition, Post Action
END			

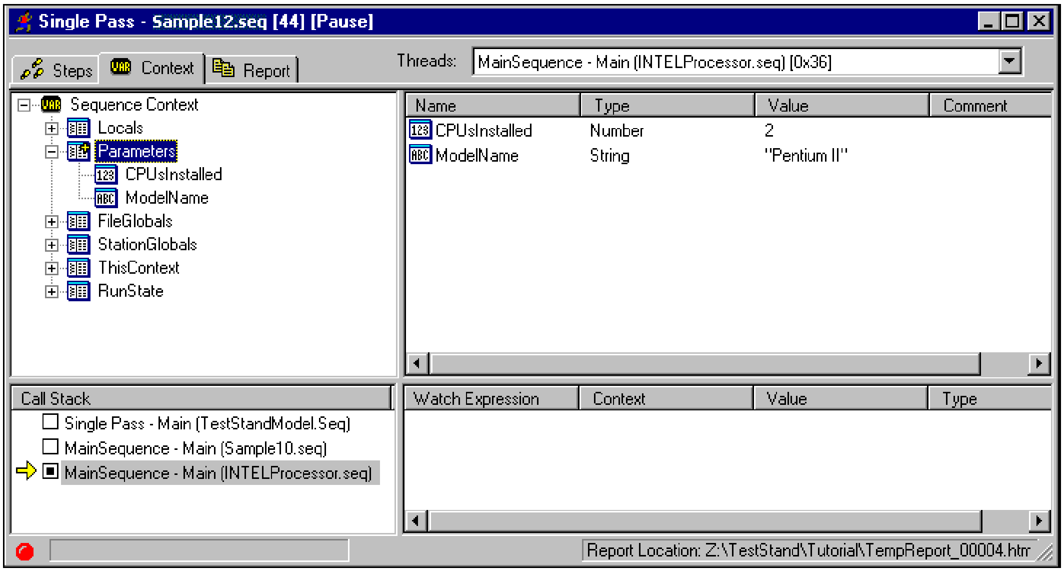
Call Stack
Single Pass - Main (SequentialModel.Seq)
MainSequence - Main (Sample10.seq)
MainSequence - Main (INTELProcessor.seq)

Watch Expression	Value	Type

Current Step Index: 0 Number of Steps: 9

Figure 11-6. INTELProcessor.seq in the Call Stack Pane

7. Click on the Context tab and notice the values of the two parameters for the sequence, as shown in Figure 11-7.



Name	Type	Value	Comment
CPUsInstalled	Number	2	
ModelName	String	"Pentium II"	

```
Single Pass - Main (TestStandModel.Seq)
MainSequence - Main (Sample10.seq)
MainSequence - Main (INTELProcessor.seq)
```

Report Location: Z:\TestStand\Tutorial\TempReport_00004.htm

Figure 11-7. Sequence Parameters in the Context Tab

The value of the `CPUsInstalled` parameter is equal to the value on the button you clicked on the Specify Number of CPUs prompt. Notice that `MainSequence` in the `INTELProcessor.seq` sequence file also requires a `ModelName` parameter. The sequence call step you created did not specify this parameter, so the engine initializes the parameter value to its default value.

8. Complete the execution by selecting **Debug»Resume**.
9. When the report file completes, review the report, but do not close the Execution window.
10. Restart the execution by selecting **Execute»Restart**.
11. Click on **Done** in the Test Simulator prompt.
12. Click on the **AMD CPU** button in the Select CPU prompt.
13. Click on the **3** button in the Number of CPUs prompt.
14. When the execution pauses at the breakpoint on the CPU Test step, step into the subsequence by selecting **Debug»Step Into**.

    Notice that the Call Stack pane lists `AMDProcessor.seq` at the bottom of the call stack.

15. Complete the execution by selecting **Debug»Resume**, and review the report.
16. Close the Execution window and the Sequence File window.

This concludes this tutorial session. In the next session, you learn how to customize the reports that TestStand generates.

12 Customizing the Report

In this chapter, you learn how to customize report generation within TestStand. Through the callback structure examined in Chapter 8, *Using Callbacks*, you can create your own Test Report Callback routine to develop reports in any format. Because changing report generation is so common, TestStand provides several options to configure the format of the test report without creating your own callback. You examine each of these options in this chapter.

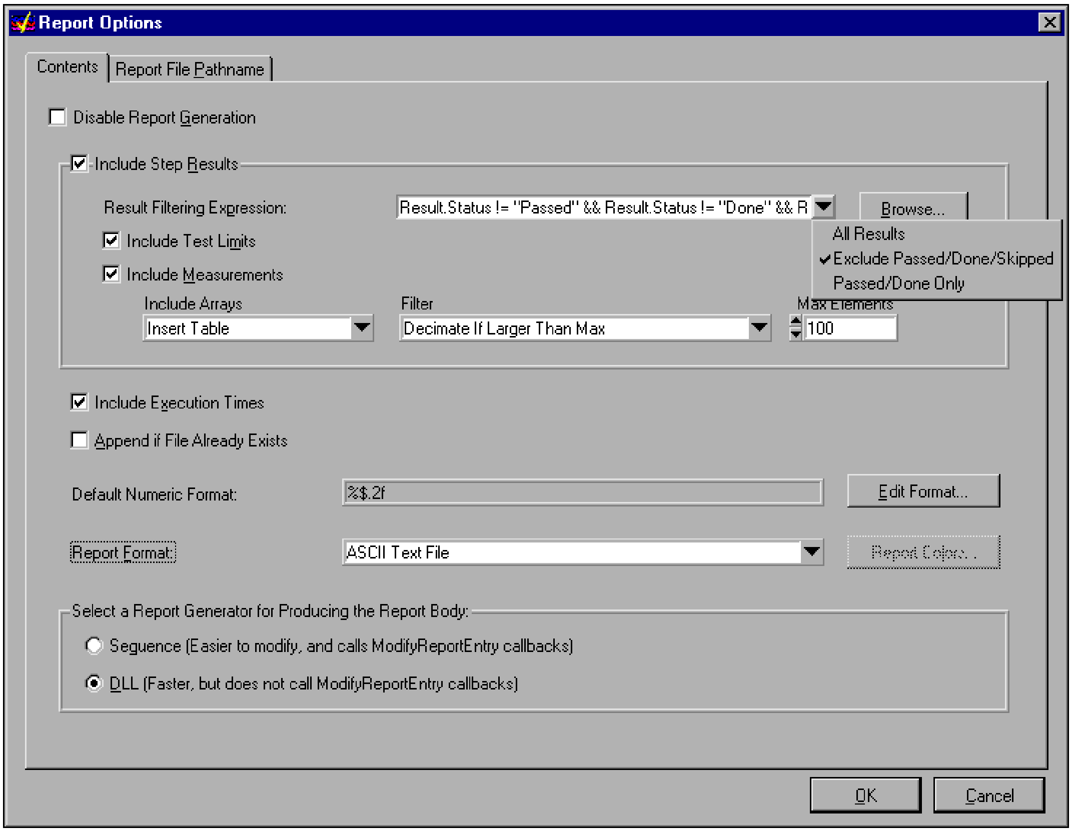
Setting Up the Example

Close all windows in the sequence editor so you can complete this tutorial session.

Configuring Test Report Options

1. Open `Sample1.seq` from the `TestStand\Tutorial` directory.

2. Select **Configure»Report Options**. The Report Settings dialog box appears, as shown in Figure 12-1.


Report Options

Contents | Report File Pathname

☐ Disable Report Generation

☑ Include Step Results

Result Filtering Expression: Result.Status != "Passed" && Result.Status != "Done" && R

All Results
✓Exclude Passed/Done/Skipped
Passed/Done Only

☑ Include Test Limits

☑ Include Measurements

Include Arrays: Insert Table

Filter: Decimate If Larger Than Max

Max Elements: 100

☑ Include Execution Times

☐ Append if File Already Exists

Default Numeric Format: %$.2f

Report Format: ASCII Text File

Select a Report Generator for Producing the Report Body:

○ Sequence (Easier to modify, and calls ModifyReportEntry callbacks)

● DLL (Faster, but does not call ModifyReportEntry callbacks)

Figure 12-1. UUT Report Setting

3. Configure the test report options as shown in Figure 12-1.

   a. Enable the Include Step Results option and configure the following Step Result settings:

      - Set the Result Filtering Expression to Exclude/Passed/Done/Skipped by clicking on the arrow to the right of the control.

        The Result Filtering Expression determines the conditions that must be met before the results are logged to the test report. In this example, you configure TestStand to record only the results of the steps that do not pass or steps that complete without any status. TestStand evaluates the expression at run-time when generating the report. The results of each step are only logged to the test report if the expression is `True`.

- Enable the Include Test Limits option.
- Enable the Include Measurements option.

  In this example, when a measurement is an array, you are configuring TestStand to include the array as a table. You can also include it as a graph if you are producing Web page reports.

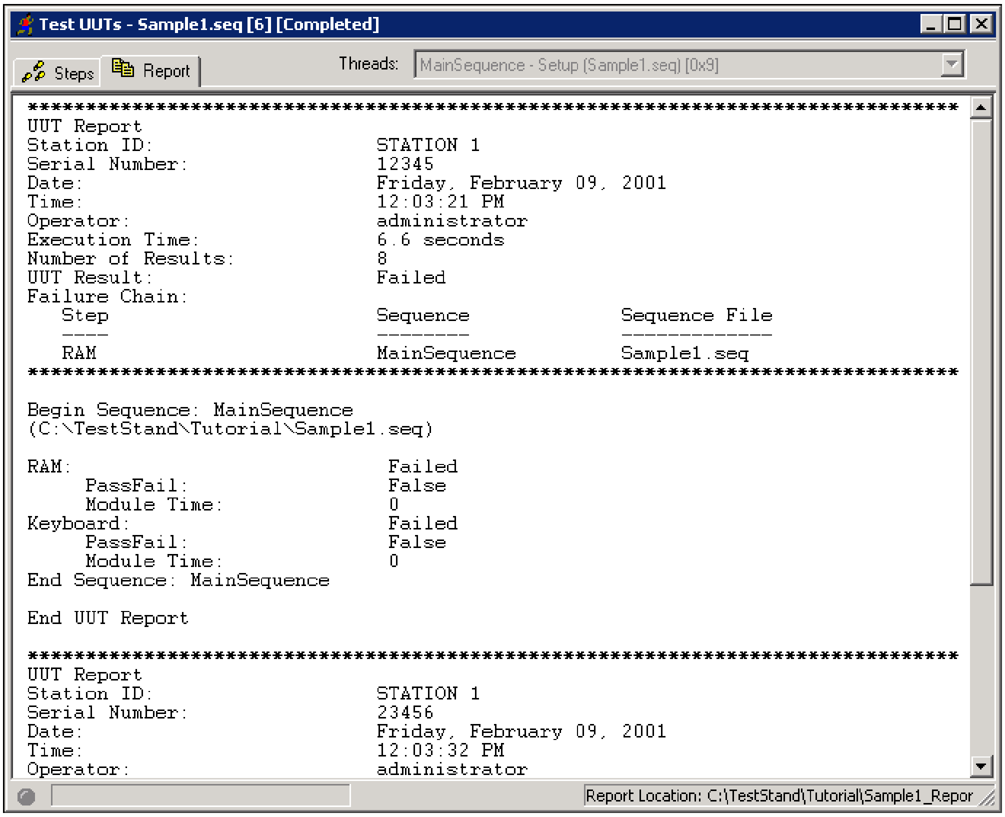
b. Enable the Include Execution Times option.

c. Click on the **Edit Format** button to display the Numeric Format dialog box. By default, TestStand configures the numeric format to report a float with 13 digits of precision. Change the Number of Fractional Digits to 2 and click the **OK** button.

d. Set the Report Format control to ASCII Text File. This setting creates the test report in a standard ASCII format.

4. Click on the Report File Pathname tab.

   This tab allows you to configure the name and path for the test report file. You can, for example, create a new file for each UUT, or include the time and date in the name of the report file.
5. Leave the Report File Pathname tab options as they are and click on **OK** to close the dialog box.
6. Execute the sequence by selecting **Execute»Test UUTs**.
7. Run through several iterations of the sequence and select different components, other than the video and CPU test, to fail.
8. Click on **Stop** in the UUT Information dialog box to stop the sequence execution.
9. Examine the test report and notice that there is a Failure Chain for UUTs that fail. The failure chain shows the step whose failure causes the UUT to fail. The failure chain also shows the sequence call steps through which the execution reaches the failing step. Figure 12-2 shows a failure chain in which the failure of the RAM step in `Sample1.seq` causes the UUT to fail. Also, notice that the only step results that appear in the report are for steps that failed. The report format should appear similar to Figure 12-2.

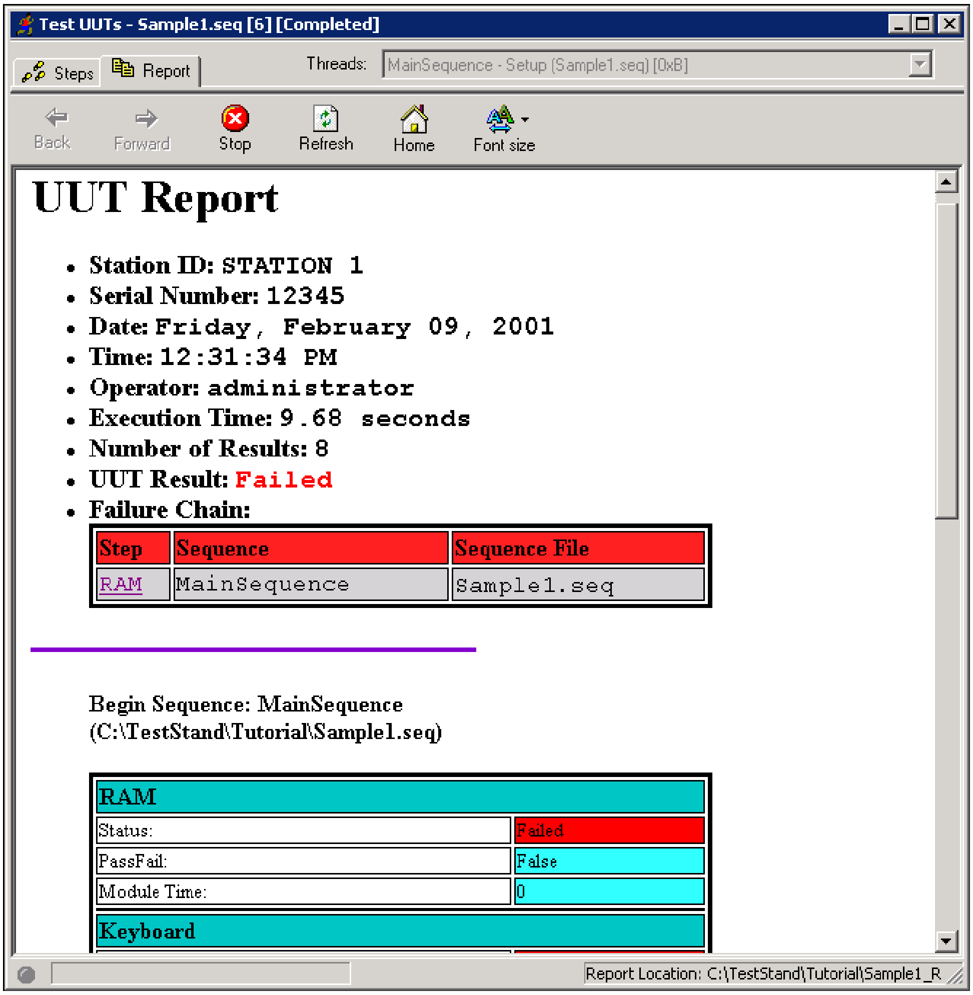

Test UUTs - Sample1.seq [6] [Completed]
Steps
Report
Threads:
MainSequence - Setup (Sample1.seq) [0x9]

```
******************************************************************************
UUT Report
Station ID:                    STATION 1
Serial Number:                 12345
Date:                          Friday, February 09, 2001
Time:                          12:03:21 PM
Operator:                      administrator
Execution Time:                6.6 seconds
Number of Results:             8
UUT Result:                    Failed
Failure Chain:
   Step                        Sequence            Sequence File
   ----                        ---------           -------------
   RAM                         MainSequence        Sample1.seq
******************************************************************************

Begin Sequence: MainSequence
(C:\TestStand\Tutorial\Sample1.seq)

RAM:                            Failed
     PassFail:                  False
     Module Time:               0
Keyboard:                       Failed
     PassFail:                  False
     Module Time:               0
End Sequence: MainSequence

End UUT Report

******************************************************************************
UUT Report
Station ID:                    STATION 1
Serial Number:                 23456
Date:                          Friday, February 09, 2001
Time:                          12:03:32 PM
Operator:                      administrator
```

Report Location: C:\TestStand\Tutorial\Sample1_Repor

Figure 12-2. Test Report in Text Format

10. Close the Execution window.
11. Select **Configure»Report Options**.
12. Change the Report Format Tag to Web Page.
13. Click on **OK** to close the dialog box.
14. Execute the sequence by selecting **Execute»Test UUTs**.
15. Run through several iterations of the sequence and select different tests, other than the Video and CPU test, to fail.
16. Click on **Stop** in the **UUT Information** dialog box to stop the sequence.
17. Examine the test report and notice that in the 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. The report format should appear as shown in Figure 12-3.


Test UUTs - Sample1.seq [6] [Completed]
Steps
Report
Threads:
MainSequence - Setup (Sample1.seq) [0xB]
Back
Forward
Stop
Refresh
Home
Font size

UUT Report

- Station ID: STATION 1
- Serial Number: 12345
- Date: Friday, February 09, 2001
- Time: 12:31:34 PM
- Operator: administrator
- Execution Time: 9.68 seconds
- Number of Results: 8
- UUT Result: Failed
- Failure Chain:

Step	Sequence	Sequence File
RAM	MainSequence	Sample1.seq

Begin Sequence: MainSequence
(C:\TestStand\Tutorial\Sample1.seq)

RAM	
Status:	Failed
PassFail:	False
Module Time:	0
Keyboard	

Report Location: C:\TestStand\Tutorial\Sample1_R

Figure 12-3. Test Report in HTML Format

18. Close the Execution window.
19. Select **Configure»Report Options**.
20. Set the Report Format control back to ASCII Text File and set the Result Filtering Expression back to `True` by selecting the All Results option in the arrow pulldown list.
21. Click on **OK** to close the Report Options dialog box.

Using External Report Viewers

You might want to view the test report in external applications more suited for displaying and editing text, such as Microsoft Word or Microsoft Excel. TestStand refers to these external applications as external report viewers.

By default the Windows 2000/NT/Me/9*x* operating systems can associate an application with a file extension. For example, by default, Microsoft associates the `.doc` file extension with the Microsoft WordPad application. If you install Microsoft Word on your system, the Word installer replaces the WordPad file type association with itself for the `.doc` file type.

1. Change the file extension of your report by completing the following steps:
   a. Select **Configure»Report Options**.
   b. Select the Report File Pathname tab.
   c. Disable the Use Standard Extension for Report Format option.
   d. Enter `doc` in the Extension string control. TestStand will now create test reports with a `.doc` file extension.
   e. Click **OK** to close the Report Options dialog box.
2. Configure TestStand to automatically launch the external viewer associated with the `.doc` file extension.
   a. Select **Configure»External Viewers**.
   b. Enable the Automatically Launch Default External Viewer checkbox.
   c. Click **OK** to close the Configure External Viewer dialog box.
3. Execute the sequence by selecting **Execute»Test UUTs**.
4. Run through several iterations of the sequence.
5. Select **Stop** in the **UUT Information** dialog box to stop the execution.

   TestStand now generates the text report and launches WordPad or Microsoft Word to display the test report.
6. Examine the test report and close the external report viewing application.
7. Change the report settings back as follows:
   a. Select **Configure»Report Options**.
   b. Change the Report Format Tag to Web Page on the Contents tab.

   c. Enable the Use Standard Extension for Report Format option on the Report File Pathname tab.
   d. Click on **OK** to close the dialog box.
8. Change the external viewer settings back as follows:
   a. Select **Configure»External Viewers**.
   b. Disable the Automatically Launch Default External Viewer dialog box.
   c. Click **OK** to close the Configure External Viewer dialog box.

If you want to define a file association independent of the operating system, you can configure this using **Configure»External Viewers**. Refer to Chapter 4, *Sequence Editor Menu Bar*, in the *TestStand User Manual* for more information on External Viewers.

Adding New Step Properties to a Report

Step types define a list of properties and behaviors for each step of that type. TestStand contains a set of predefined step types. If the functionality of these step types does not meet your needs, you can design ones that do.

In this exercise you create a new step type with a custom numeric array property and include this property in the report.

Setting Up the Example

Close all windows in the sequence editor so you can complete this example.

Creating a Step Type

1. Open a new sequence by selecting **File»New Sequence File**.
2. Save the sequence by selecting **File»Save As**. Save the sequence as `Sample13.seq` in the `<TestStand>\Tutorial` directory. By saving the sequence file now, you can specify relative paths to code modules instead of absolute paths.
3. Choose Sequence File Types from the View ring control in the Sequence File window. As shown in Figure 12-4, by default, the Step Types tab should be selected.
4. Right-click and select **Insert Step Type** from the context menu. Rename the step type `NumericArray`.

5. Insert a custom step property that is a numeric array. In the left tree view pane expand the NumericArray and highlight the Result node. During execution, TestStand automatically collects the values of all properties within the Result container of a step. It stores these as elements of the Locals.ResultList array. These values can be included in the report, as described below. In the right list view pane right-click on the Error property and select **Insert Field»Array of»Number** from the context menu, as shown in Figure 12-4.

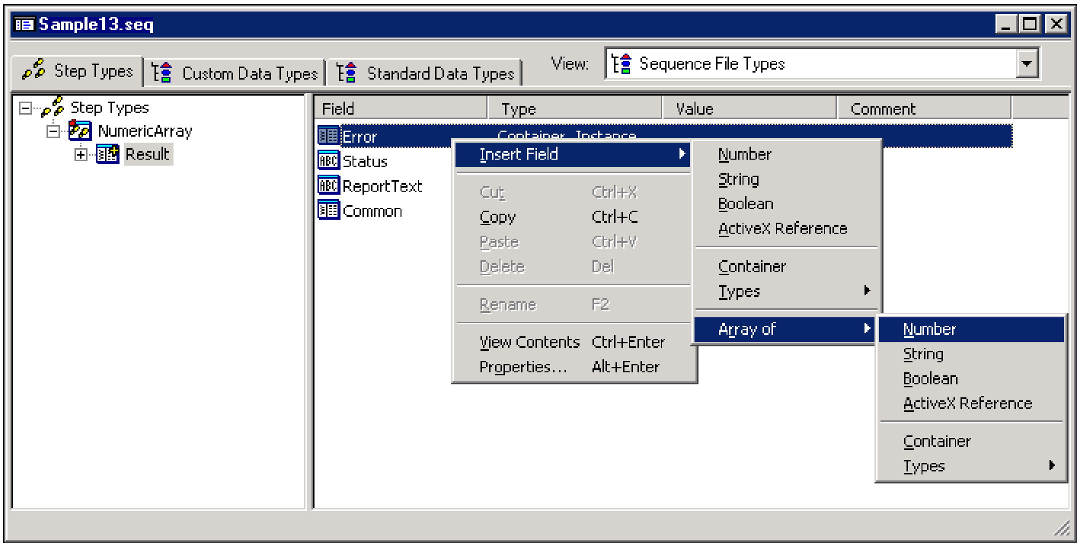


Figure 12-4. Insert Numeric Array Context Menu

6. In the Array Bounds dialog box of the inserted property, enable the Initially Empty option and click the **OK** button. Rename the property `NumArray`. When you write an array to this property, TestStand automatically resizes the array to the size of your one-dimensional array.

7. Right-click on the NumArray property and select **Properties** from the context menu, as shown in Figure 12-5.

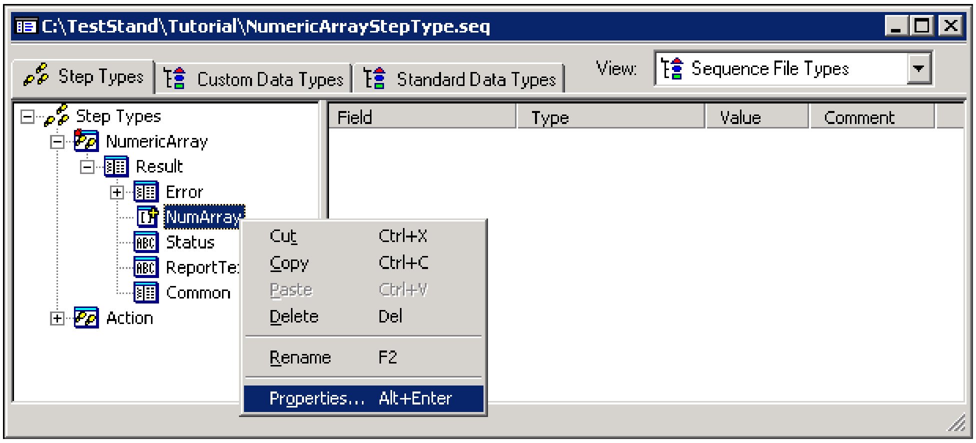


Figure 12-5. Numeric Array Properties Context Menu

8. Click on the **Advanced** button in the NumArray Properties dialog box.
9. In the Edit Flags dialog box, enable PropFlags_IncludeInReport from the list of available flags. This flag informs TestStand to add the value of the property to the report.
10. Click the **OK** button twice to close the Edit Flags dialog box and the NumArray Properties dialog box.
11. Right-click on the NumericArray step type and choose **Properties** from the context menu. The NumericArray Properties dialog box allows you to configure the behavior of your step type including the default values of all step properties such as run options, looping options, expressions, and post actions. You can create substeps that call code modules before and after a user-defined step module. Substeps define standard actions that occur for every instance of a step type. For more information about step types refer to Chapter 9, *Types*, of the *TestStand User Manual*.

12. Under the General tab, enter the following values into the corresponding controls.

Control Name	Control Value
Default Step Name Expression	“Numeric Array Step”
Step Description Expression	“My Description” + “%ModuleDescription”

    Leave the other controls set to their default values.

13. Click the **OK** button to close the NumericArray Properties dialog box. You will not modify any other step type settings.
14. Save your sequence file by selecting **File»Save**.

You have created a step type with a property that can receive a numeric array from a step module. By locating this property within the Result container of the step type and enabling its PropFlags_IncludeInReport flag, the numeric array for each instance of your step type is included in the report.

There are other step type capabilities that you do not use in this exercise. Typical modifications of a step type might include one or more post substeps or a status expression that evaluates the numeric array to determine whether the step passes or fails. The step type might also have one or more edit substeps that allow the operator to configure the pass/fail criteria prior to executing the test sequence. In an instance of a step type a context menu item is created for each edit substep. You also might create code templates to help users create code modules that return an array.

Creating a Step Module Using the LabVIEW Standard Prototype Adapter

In this exercise, you create an instance of your step type and a VI step module that returns a numeric array. If you are not using LabVIEW, but you do use LabWindows/CVI, you can skip this section and proceed to the *Creating a Step Module Using the C/CVI Standard Prototype Adapter* or the *Creating a Step Module Using the DLL Flexible Prototype Adapter* section in this chapter.

1. Open `<TestStand>\Tutorial\Sample13.seq`.
2. Select MainSequence from the View ring control.
3. Click on the Adapter Selection ring on the toolbar and select the LabVIEW Standard Prototype Adapter.

4. In the Main step group of your MainSequence, right-click and select **Insert Step»NumericArray** from the context menu. Notice that the default step name and step description are the values you entered when you created the step type.
5. Right-click on the Numeric Array Step and select **Specify Module** from the context menu.
6. Enable the Sequence Context ActiveX Pointer option on the Edit LabVIEW VI Call dialog box.
7. Click on the **Create VI** button and TestStand prompts you to select a pathname for the step's code module.
8. Find the `<TestStand>\Tutorial` directory. Type the name `ReturnNumArray.vi` in the File name control. The VI might already exist if someone else previously completed this session of the tutorial.
9. Click on **OK** to close the dialog box you used to select the VI pathname.
10. TestStand creates and opens the `ReturnNumArray.vi` in LabVIEW. Complete the diagram of the VI, as shown in Figure 12-6. A common mistake is to type a syntax error in the lookup string `Step.Result.NumArray`.

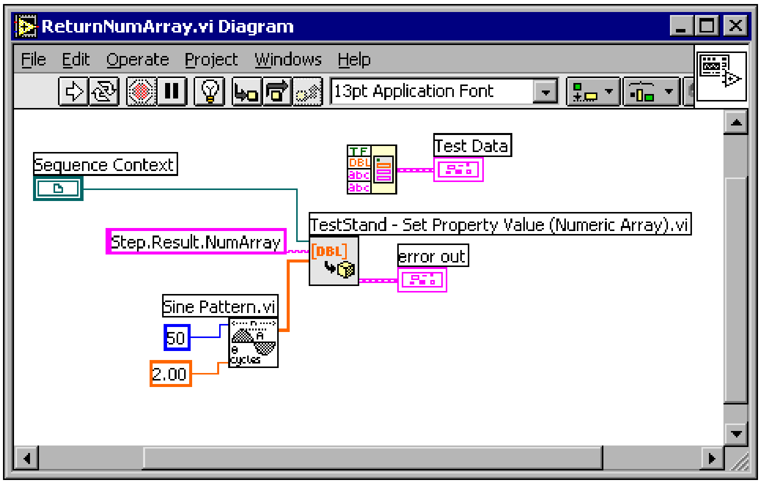
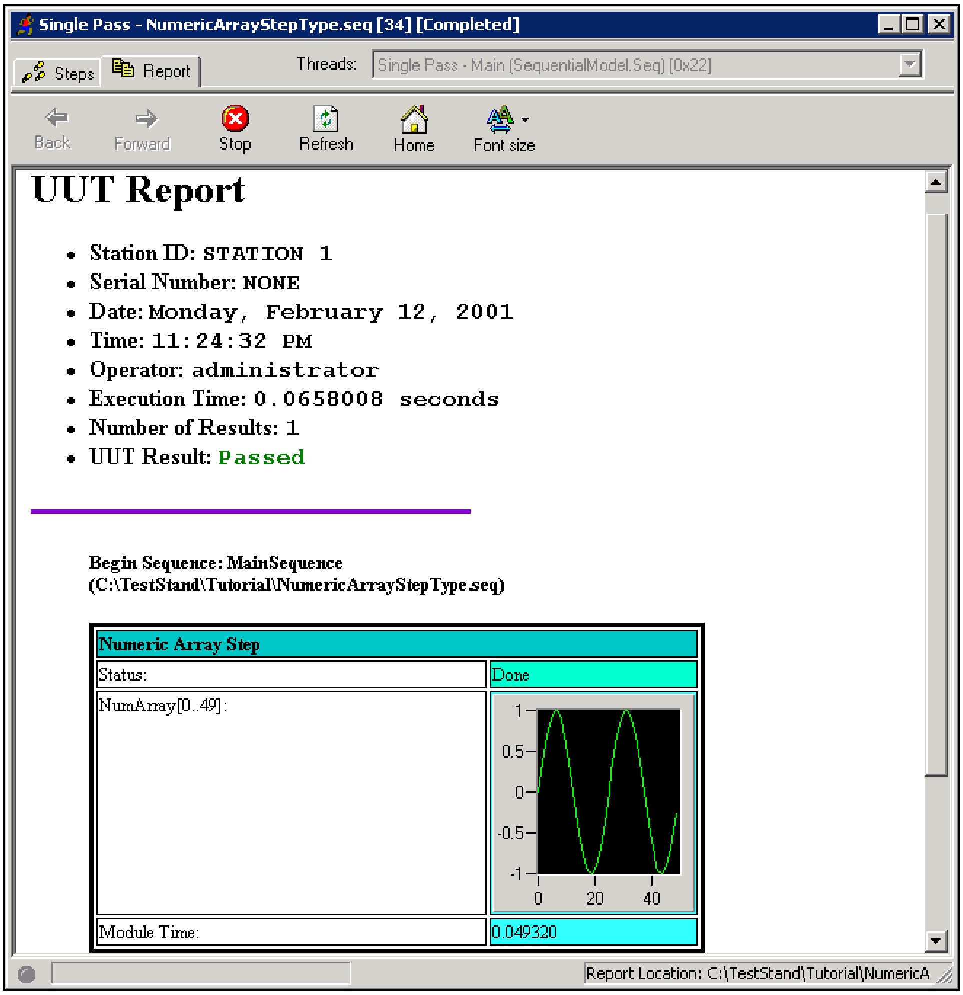


Figure 12-6. ReturnNumArray.vi Diagram

11. After you finish building the VI, save it by selecting **File»Save** in LabVIEW.
12. Close the VI diagram and front panel.

13. Return to the sequence editor, and close the Edit LabVIEW VI Call dialog box by clicking on **OK**.
14. Save your sequence file by selecting **File»Save As**.
15. Execute the sequence by selecting **Execute»Single Pass**. If you have configured the Report Options so that the Report Format control is Web Page and the Include Arrays control is Insert Graph, then the array data should appear as shown in Figure 12-7. Vary these report option control values to see how they affect the report.


Single Pass - NumericArrayStepType.seq [34] [Completed]
Steps
Report
Threads:
Single Pass - Main (SequentialModel.Seq) [0x22]
Back
Forward
Stop
Refresh
Home
Font size
1
0.5
0
-0.5
-1
0
20
40
Report Location: C:\TestStand\Tutorial\NumericA

UUT Report

- Station ID: STATION 1
- Serial Number: NONE
- Date: Monday, February 12, 2001
- Time: 11:24:32 PM
- Operator: administrator
- Execution Time: 0.0658008 seconds
- Number of Results: 1
- UUT Result: Passed

Begin Sequence: MainSequence
(C:\TestStand\Tutorial\NumericArrayStepType.seq)

Numeric Array Step	
Status:	Done
NumArray[0..49]:	
Module Time:	0.049320

Figure 12-7. Numeric Array Report with Graph

Creating a Step Module Using the C/CVI Standard Prototype Adapter

In this exercise, you create an instance of your step type and a DLL code module that returns a numeric array. If you use LabVIEW you can skip this section and proceed to Chapter 13, *Converting LabVIEW and LabWindows/CVI Test Executive Sequences*.

1. Open `<TestStand>\Tutorial\Sample13.seq`.
2. Select MainSequence from the View ring control.
3. Click on the Adapter Selection ring on the toolbar and select the C/CVI Standard Prototype Adapter.
4. In the Main step group of your MainSequence, right-click and select **Insert Step»NumericArray** from the context menu. Notice that the default step name and step description are the values you entered when you created the step type.
5. Right-click on the Numeric Array Step and select **Specify Module** from the context menu. Under the Module tab of the Edit C/CVI Module Call dialog box, the Module Type should be set to Dynamic Link Library. Type `NumericArray.dll` into the Module Pathname control.
6. Type `GetNumArray` into the Function Name control.
7. Enable the Pass Sequence Context option on the Edit C/CVI Module Call dialog box.
8. Under the Source Control tab of the Edit C/CVI Module Call dialog box, type `NumericArray.c` into the Pathname of Source File Containing Function control.
9. Type `NumericArray.prj` into the Pathname of CVI Project to Open control.
10. Click on the **Create Code** button. When you make this selection, TestStand prompts you to select a pathname for the LabWindows/CVI project file.
11. Find the `<TestStand>\Tutorial` directory and click the **OK** button to close the Select a Pathname for the CVI Project File dialog box. The file might already exist if someone else previously completed this session of the tutorial.
12. TestStand prompts you to select a pathname for the LabWindows/CVI source file. Find the `<TestStand>\Tutorial` directory and click the **OK** button to close the Select a Pathname for the Source File dialog box.

After you enter the name, TestStand does the following:

a. Launches an external instance of LabWindows/CVI
b. Creates a new project file in LabWindows/CVI
c. Creates the source file
d. Adds the source file and the TestStand support instrument drivers to the project
e. Generates a template `GetNumArray` function in the source file

13. Update the `GetNumArray` function to return an array of doubles to the Step.Result.NumArray property. Update the source code for the function as follows. Changed lines appear in bold.

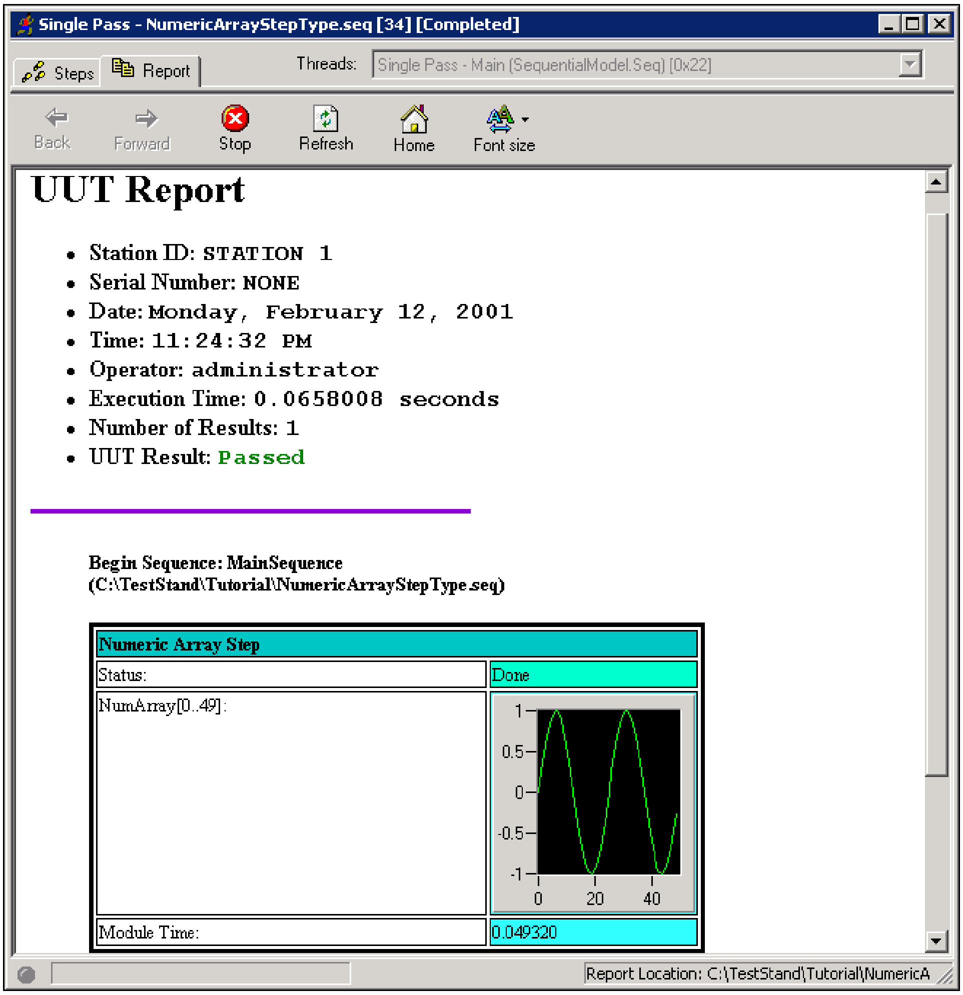
```
void __declspec(dllexport) TX_TEST
   GetNumArray(tTestData *testData, tTestError
   *testError)
{
   int                  error = 0;
   ErrMsg               errMsg = {'\0'};
   ERRORINFO            errorInfo;
   VARIANT              tmpVariant;
   double               sineArray[50];

   tmpVariant = CA_VariantEmpty();
   errChk(SinePattern (50, 1.0, 0.0, 2, sineArray));
   errChk(CA_VariantClear(&tmpVariant));

   // Create a Safe Array from the 1-D array, and
   // store the Safe Array in the VARIANT.
   errChk(CA_VariantSet1DArray(&tmpVariant,
      CAVT_DOUBLE, 50, sineArray));

   // Set the value of the property the lookupString
   // parameter specifies with a variant.
   // Use this method to set the value
   // of an entire array at once.
   tsErrChk(TS_PropertySetValVariant(testData->
      seqContextCVI, &errorInfo,
      "Step.Result.NumArray", 0, tmpVariant));

Error:
   // FREE RESOURCES
   CA_VariantClear(&tmpVariant);
```

```
    // If an error occurred, set the error flag to
    // cause a run-time error in TestStand.
    if (error < 0)
        {
        testError->errorFlag = TRUE;
        testError->errorCode = error;
        testData->replaceStringFuncPtr(&testError->
        errorMessage, errMsg);
        }
    return;
}
```

14. Compile the source code by selecting **Build»Compile File** to verify that your changes are correct.
15. Save the source code after you successfully compile.
16. To build the DLL, select **Build»Create Debuggable Dynamic Link Library** in the Project window.

Note If LabWindows/CVI returns a file permission error when you create the DLL, return to the sequence editor and select **File»Unload All Modules**. When you make this selection, TestStand unloads all step code modules, which includes DLLs, VIs and any other modules the adapter loads. Return to LabWindows/CVI and rebuild the DLL.

17. After the DLL build is complete, return to the sequence editor.
18. Click on **OK** to close the Edit C/CVI Module Call dialog box and return to the Main step group view.
19. Save your sequence file by selecting **File»Save As**.
20. Execute the sequence by selecting **Execute»Single Pass**. If you have configured the Report Options so that the Report Format control is Web Page and the Include Arrays control is Insert Graph, then the array data should appear as shown in Figure 12-8. Vary these report option control values to see how they affect the report.


Single Pass - NumericArrayStepType.seq [34] [Completed]
Steps
Report
Threads:
Single Pass - Main (SequentialModel.Seq) [0x22]
Back
Forward
Stop
Refresh
Home
Font size
Report Location: C:\TestStand\Tutorial\NumericA

UUT Report

- Station ID: STATION 1
- Serial Number: NONE
- Date: Monday, February 12, 2001
- Time: 11:24:32 PM
- Operator: administrator
- Execution Time: 0.0658008 seconds
- Number of Results: 1
- UUT Result: Passed

Begin Sequence: MainSequence
(C:\TestStand\Tutorial\NumericArrayStepType.seq)

Numeric Array Step	
Status:	Done
NumArray[0..49]:	
Module Time:	0.049320

Figure 12-8. Numeric Array Report with Graph

Creating a Step Module Using the DLL Flexible Prototype Adapter

In this exercise, you create an instance of your step type and a DLL step module that returns a numeric array. If you use LabVIEW you can skip this section and proceed to Chapter 13, *Converting LabVIEW and LabWindows/CVI Test Executive Sequences*.

1. Open `<TestStand>\Tutorial\Sample13.seq`.
2. Select MainSequence from the View ring control.

3. Click on the Adapter Selection ring on the toolbar and select the DLL Flexible Prototype Adapter.
4. In the Main step group of your MainSequence, right-click and select **Insert Step»NumericArray** from the context menu. Notice that the default step name and step description are the values you entered when you created the step type.
5. Right-click on the Numeric Array Step and select **Specify Module** from the context menu. Under the Module tab of the Edit DLL Call dialog box, type `NumericArray.dll` into the Module Pathname control.
6. Type `GetNumericArray` into the Function Name control.
7. The value of the Calling Convention control should remain Standard Call.
8. Under the Source Code tab, type `NumericArray.c` in the Pathname of Source File Containing Function control.
9. Click the **Create Code** button.
10. If the source file does not already exist, TestStand prompts you to choose a pathname for the source file. Browse to the `<TestStand>/Tutorial` directory and click the **OK** button.
11. TestStand either opens the `.c` file with an application on your system that is registered to open files with a `*.c` extension, or it prompts you as to whether you want to launch Notepad to view the newly created `.c` file. After viewing the `.c` file, return to the sequence editor and click the **OK** button to close the Edit DLL Call dialog box.
12. Launch LabWindows/CVI by selecting **Start»Programs»National Instruments»Measurement Studio»CVI IDE**.
13. If you completed the previous exercise, open the LabWindows/CVI project `NumericArray.prj`.
14. If you did not complete the previous exercise, you need to create a LabWindows/CVI project with which to build a DLL module.
    a. Open a new project by selecting **File»New»Project**. If LabWindows/CVI already has a project loaded, LabWindows/CVI prompts you as to whether you want to unload the current project. Click the **Yes** button. LabWindows/CVI prompts you as to whether to transfer the current project options to the new project. Uncheck all options and click on the **OK** button.

b. Select **Edit»Add Files to Project»All Files (*.*)** to open the Add Files to Project dialog box. Browse to each of the files below and use the Add button to add them to the Selected Files control.

   - `<TestStand>\Tutorial\ NumericArray.c`
   - `<TestStand>\API\CVI\tsapicvi.fp`
   - `<TestStand>\API\CVI\tsutil.fp`

   After adding all files click the **OK** button to return to the project window where the file names should be listed.

c. Save the project as `NumericArray.prj` in the `<TestStand>\Tutorial` directory by selecting **File»Save**.

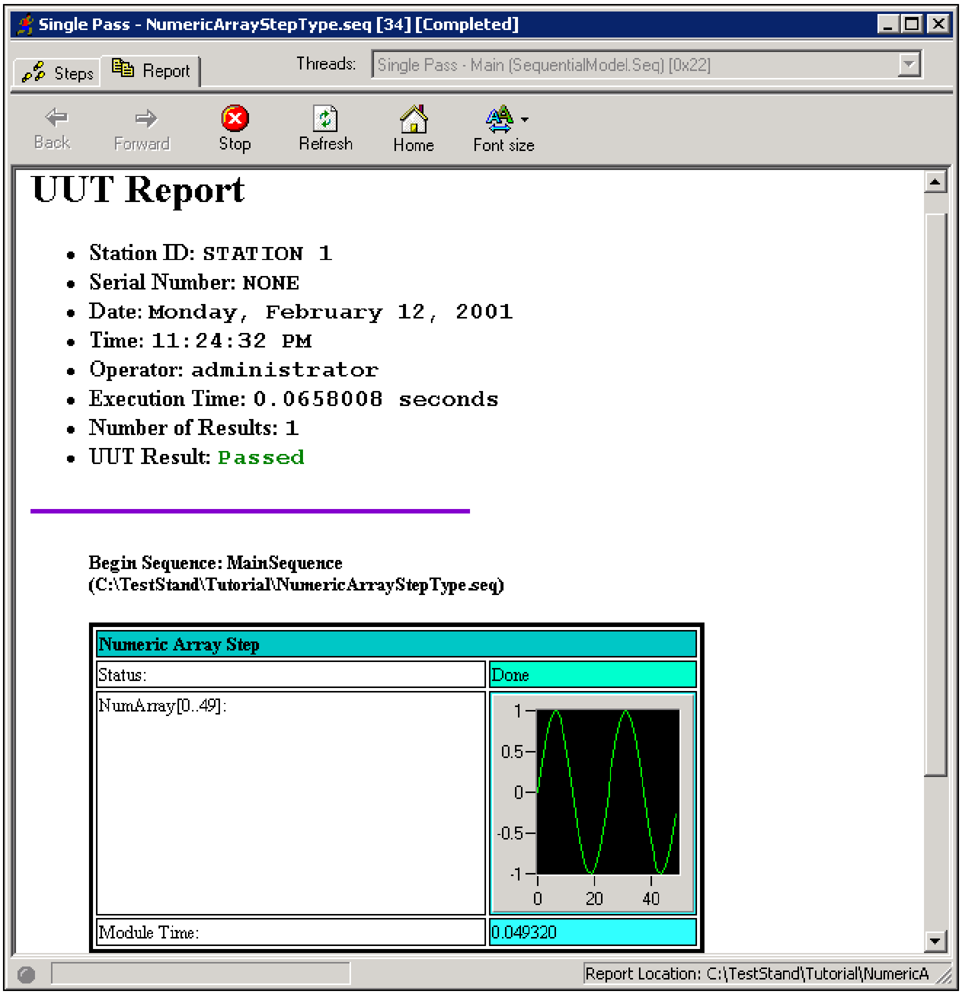
15. Open the `NumericArray.c` file by double clicking on the name of the file in the project window.
16. Update the `GetNumericArray` function to return an array of doubles to the Step.Result.NumArray property. Update the source code for the function as follows. Changed lines appear in bold.

```
void __declspec(dllexport) __stdcall GetNumericArray
   (CAObjHandle seqContextCVI, short *errorOccurred,
   long *errorCode, char errorMsg[1024])
{
int         error = 0;
ErrMsg      errMsg = {'\0'};
ERRORINFO   errorInfo;
VARIANT     tmpVariant;
double      sineArray[50];

tmpVariant = CA_VariantEmpty();
errChk(SinePattern (50, 1.0, 0.0, 2, sineArray));
errChk(CA_VariantClear(&tmpVariant));

// Create a Safe Array from the 1-D array,
// and stores the Safe Array in the VARIANT.
errChk(CA_VariantSet1DArray(&tmpVariant,
   CAVT_DOUBLE, 50, sineArray));


// Set the value of the property the lookupString
// parameter specifies with a variant.
// Use this method to set the value
// of an entire array at once.
tsErrChk(TS_PropertySetValVariant(seqContextCVI,
            &errorInfo, "Step.Result.NumArray", 0,
            tmpVariant));

Error:
    // FREE RESOURCES
    CA_VariantClear(&tmpVariant);

    // If an error occurred, set the error flag to cause
    // a run-time error in TestStand.
    if (error < 0)
        {
        testError->errorFlag = TRUE;
        *errorCode = error;
        strcpy(errorMsg, errMsg);
        }
    return;
}
```

17. Compile the source code by selecting **Build»Compile File** to verify that your changes are correct.
18. Save the source code after you successfully compile.
19. From the Project window select **Build»Target Type»Dynamic Link Library** so that the project builds a DLL.
20. Build the DLL by selecting **Build»Create Debuggable Dynamic Link Library**.

Note If LabWindows/CVI returns a file permission error when you create the DLL, return to the sequence editor and select **File»Unload All Modules**. When you make this selection, TestStand unloads all step code modules, which includes DLLs, VIs and any other modules the adapter loads. Return to LabWindows/CVI and rebuild the DLL.

21. Return to the sequence editor and save your sequence file by selecting **File»Save**.
22. You are now ready to execute the sequence that calls your DLL function. Execute the sequence by selecting **Execute»Single Pass**.
23. If you have configured the Report Options so that the Report Format control is Web Page and the Include Arrays control is Insert Graph, then the array data should appear as shown in Figure 12-9. Vary these report option control values to see how they affect the report.



Single Pass - NumericArrayStepType.seq [34] [Completed]

Threads: Single Pass - Main (SequentialModel.Seq) [0x22]

UUT Report

- Station ID: STATION 1
- Serial Number: NONE
- Date: Monday, February 12, 2001
- Time: 11:24:32 PM
- Operator: administrator
- Execution Time: 0.0658008 seconds
- Number of Results: 1
- UUT Result: Passed

Begin Sequence: MainSequence
(C:\TestStand\Tutorial\NumericArrayStepType.seq)

Numeric Array Step	
Status:	Done
NumArray[0..49]:	
Module Time:	0.049320

Report Location: C:\TestStand\Tutorial\NumericA

Figure 12-9. Numeric Array Report with Graph

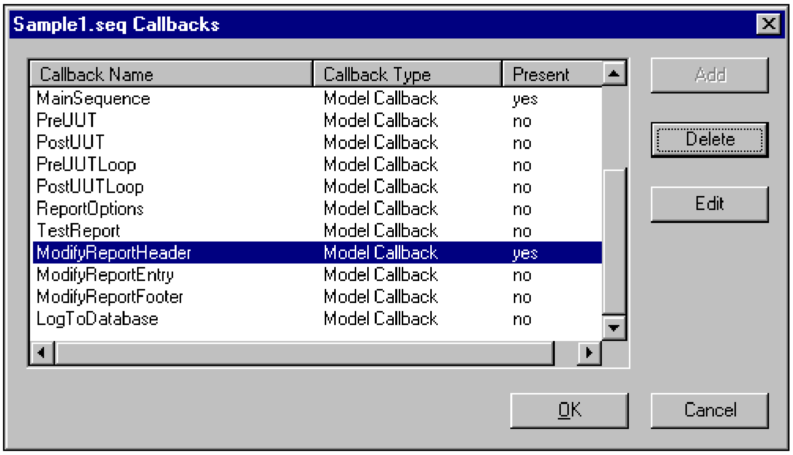
In this exercise, you used a sequence context pointer to pass the numeric array from your DLL to TestStand. Using the Flexible DLL Prototype adapter you also have the option of passing a numeric array directly using the function parameters. For more information on how to do this, refer to the example, `<TestStand>\Examples\AccessingArrays\PassingArrayParametersToDLL`.

Adding to a Report Using Callbacks

The default header for an HTML report appears as shown in Figure 12-3. In this exercise, you add a logo to the header of the HTML report using a report callback in the process model.

1. Open `<TestStand>\Tutorial\Sample1.seq`.
2. With the Sequence File window as the active window, select **Edit»Sequence File Callbacks** to open the `Sample1.seq` Callbacks dialog box.
3. Select the `ModifyReportHeader` callback.
4. Click on the **Add** button to add a callback to the sequence file.

   Figure 12-10 shows the resulting dialog box.



Callback Name	Callback Type	Present
MainSequence	Model Callback	yes
PreUUT	Model Callback	no
PostUUT	Model Callback	no
PreUUTLoop	Model Callback	no
PostUUTLoop	Model Callback	no
ReportOptions	Model Callback	no
TestReport	Model Callback	no
ModifyReportHeader	Model Callback	yes
ModifyReportEntry	Model Callback	no
ModifyReportFooter	Model Callback	no
LogToDatabase	Model Callback	no

Figure 12-10. Callbacks Dialog Box

5. Click on the **Edit** button to close the dialog box and edit the new `ModifyReportHeader` callback sequence. Notice that the View pull-down ring lists the `ModifyReportHeader` as the active sequence, as shown in Figure 12-11.

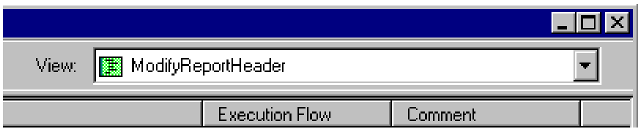


Figure 12-11. ModifyReportHeader in Sequence View

6. Click on the Locals tab and right-click in the right pane to insert a local string variable, as shown in Figure 12-12.

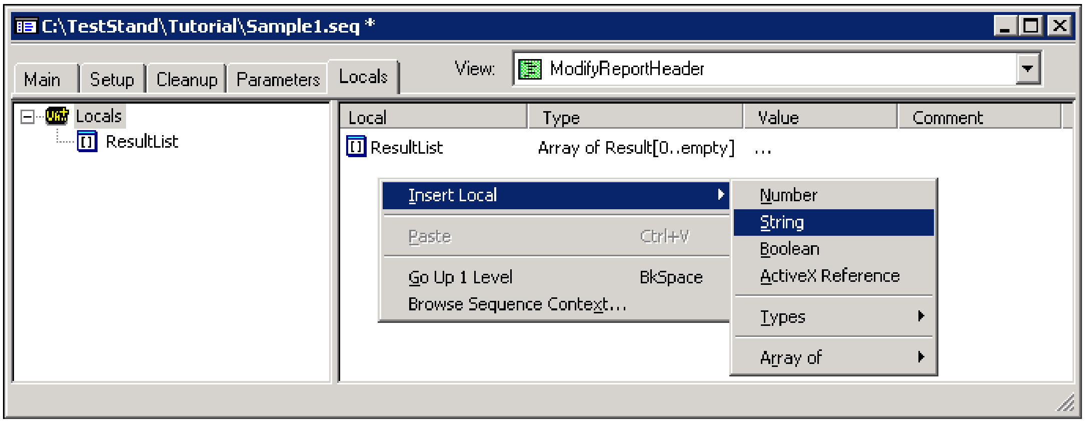


Figure 12-12. Insert String Local in ModifyReportHeader

7. Rename the variable `AddToHeader`.
8. Double-click on AddToHeader.
9. Enter the following value in the Value field of the String Properties dialog box:

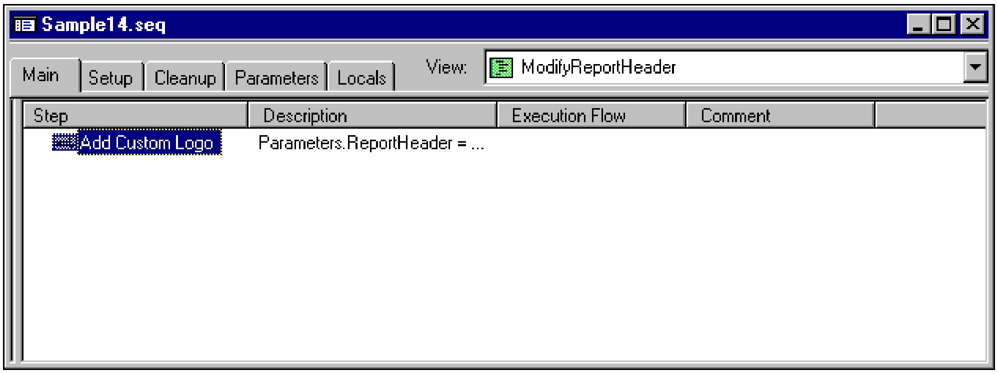
```
<IMG ALT='Logo Goes Here' SRC='Logo.jpg'><br><br>
<A HREF='http://www.ni.com'>Visit Our Web
  Site</A><br>
```

10. Click on **OK** to close the dialog box.
11. Click on the Main tab to display the Main step group, which is empty.
12. Right-click in the steps list and insert a Statement step by selecting **Insert Step»Statement** in the context menu.
13. Rename this step `Add Custom Logo`.
14. Right-click on the Add Custom Logo step and select **Edit Expression** from the context menu.
15. Enter the following expression:

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

16. Click on **OK** to close the Edit Statement Step dialog box.
17. Select **File»Save As** and save the sequence in the `TestStand\Tutorial` directory as `Sample14.seq`.

    Figure 12-13 shows the resulting sequence file.

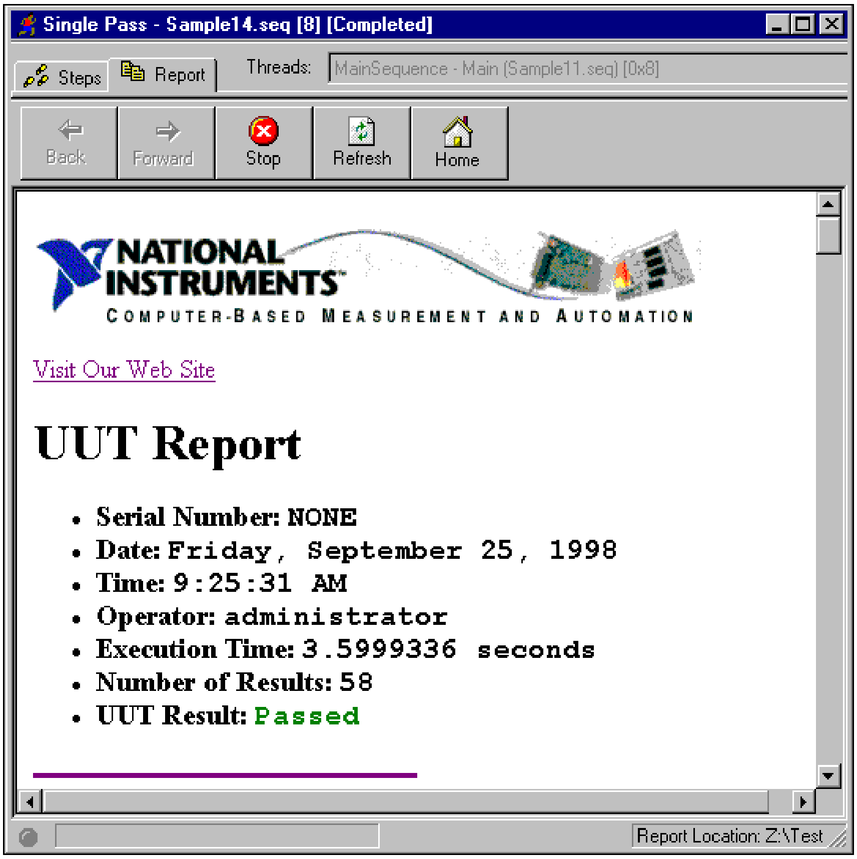


Step	Description	Execution Flow	Comment
Add Custom Logo	Parameters.ReportHeader = ...		

Figure 12-13. Completed ModifyReportHeader Sequence

18. Select **Execute»Single Pass**.
19. Click on **Done** in the Test Simulator prompt.

20. After the execution completes, view the report and notice the new logo image at the top of the UUT Report, as shown in Figure 12-14.


Single Pass - Sample14.seq [8] [Completed]
Steps
Report
Threads:
MainSequence - Main (Sample11.seq) [0x8]
Back
Forward
Stop
Refresh
Home
NATIONAL INSTRUMENTS
COMPUTER-BASED MEASUREMENT AND AUTOMATION
Visit Our Web Site
Report Location: Z:\Test

UUT Report

- Serial Number: NONE
- Date: Friday, September 25, 1998
- Time: 9:25:31 AM
- Operator: administrator
- Execution Time: 3.5999336 seconds
- Number of Results: 58
- UUT Result: Passed

Figure 12-14. New HTML Header

21. Close the Execution window and the Sequence File window.

This concludes this final session in this tutorial. For more details on customizing reports, refer to Chapter 15, *Managing Reports*, in the *TestStand User Manual*.

13 Converting LabVIEW and LabWindows/CVI Test Executive Sequences

This chapter explains how to convert existing test sequences from Test Executive to TestStand. TestStand provides a conversion utility for converting LabVIEW Test Executive and LabWindows/CVI Test Executive sequence files to TestStand sequence files.

Converting LabVIEW Test Executive Sequences

To convert a LabVIEW Test Executive 5.1 sequence file to a TestStand sequence file, complete the following steps:

1. Select **Tools»Sequence File Converters»Convert LabVIEW Test Executive Sequence** in either the TestStand Sequence Editor or any of the execution interfaces.
2. In the file dialog box that appears, select a LabVIEW Test Executive sequence file to convert.
3. Enter a new file name for the converted test sequence.

   TestStand displays a message indicating whether the conversion was successful.

Refer to the online help document in the `TestStand\Doc` directory, *Converting from the LabVIEW Test Executive to TestStand* (`LVTECompatibility.hlp`), for more information on converting sequences.

Note If you are using version 4.0 or 5.0 of the LabVIEW Test Executive, you must first convert your sequence files to version 5.1 before switching to TestStand.

Converting LabWindows/CVI Test Executive Sequences

To convert a LabWindows/CVI Test Executive 2.0.1 sequence file to a TestStand sequence file, complete the following steps:

1. Select **Tools»Sequence File Converters»Convert CVI Test Executive Sequence** in either the TestStand Sequence Editor or any of the run-time operator interfaces.
2. In the file dialog box that appears, select a LabWindows/CVI Test Executive sequence file to convert.
3. Enter a new file name for the converted test sequence.

   TestStand displays a message indicating whether the conversion was successful.

Refer to the online help document in the `TestStand\Doc` directory, *Converting from the LabWindows/CVI Test Executive to TestStand* (`CVITECompatibility.hlp`), for more information on converting sequences.

Note If the version of your LabWindows/CVI Test Executive is earlier than 2.0.1, you must first load and re-save the sequence file in the sequence editor for LabWindows/CVI Test Executive 2.0.1.

A

Technical Support Resources

Web Support

National Instruments Web support is your first stop for help in solving installation, configuration, and application problems and questions. Online problem-solving and diagnostic resources include frequently asked questions, knowledge bases, product-specific troubleshooting wizards, manuals, drivers, software updates, and more. Web support is available through the Technical Support section of `ni.com`

NI Developer Zone

The NI Developer Zone at `ni.com/zone` is the essential resource for building measurement and automation systems. At the NI Developer Zone, you can easily access the latest example programs, system configurators, tutorials, technical news, as well as a community of developers ready to share their own techniques.

Customer Education

National Instruments provides a number of alternatives to satisfy your training needs, from self-paced tutorials, videos, and interactive CDs to instructor-led hands-on courses at locations around the world. Visit the Customer Education section of `ni.com` for online course schedules, syllabi, training centers, and class registration.

System Integration

If you have time constraints, limited in-house technical resources, or other dilemmas, you may prefer to employ consulting or system integration services. You can rely on the expertise available through our worldwide network of Alliance Program members. To find out more about our Alliance system integration solutions, visit the System Integration section of `ni.com`

Worldwide Support

National Instruments has offices located around the world to help address your support needs. You can access our branch office Web sites from the Worldwide Offices section of `ni.com`. Branch office Web sites provide up-to-date contact information, support phone numbers, e-mail addresses, and current events.

If you have searched the technical support resources on our Web site and still cannot find the answers you need, contact your local office or National Instruments corporate. Phone numbers for our worldwide offices are listed at the front of this manual.

Glossary

Prefix	Meaning	Value
p-	pico-	10^{-12}
n-	nano-	10^{-9}
µ-	micro-	10^{-6}
m-	milli-	10^{-3}
k-	kilo-	10^{3}
M-	mega-	10^{6}
G-	giga-	10^{9}
t-	tera-	10^{12}

A

abort	To stop an execution without running any of the Cleanup step groups in the sequences on the call stack run. When you abort an execution, no report generation occurs.
active window	The window that user input affects at a given moment. The title of an active window is highlighted.
ActiveX reference property	A container of information that maintains a reference to an ActiveX object. TestStand maintains the value of the property as an `IDispatch` or `IUnknown` pointer.
ActiveX server	Any executable code that makes itself available to other applications according to the ActiveX standard. ActiveX implies a client/server relationship in which the client requests objects from the server and asks the server to perform actions on the objects.
Adapter	A service of the TestStand engine that steps use to invoke code in another sequence or in a code module. The adapter knows the type of the code module, how to call it, and how to pass parameters to it.
administrator	A user profile that usually contains all privileges for a test station.

Application Development Environment (ADE)	A programming environment such as LabVIEW, LabWindows/CVI, or Microsoft Visual C, in which you can create test modules and run-time execution operator interfaces.
Application Programming Interface (API)	A set of classes, methods, and properties that you use to control a specific service, such as the TestStand engine.
array property	A property that contains an array of single-valued properties of the same type.
ASCII	American Standard Code for Information Interchange.

B

block diagram	Pictorial description or representation of a program or algorithm. In LabVIEW, the block diagram which consists of executable icons called nodes and wires that carry data between the nodes, is the source code for the VI. The block diagram resides in the Diagram window of the VI.
breakpoint	An interruption in the execution of a program.
built-in property	A property that all steps or sequences contain. An example is the step run mode property. TestStand normally hides these properties in the sequence editor, although it lets you modify 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

call stack	The chain of active sequences that are waiting for nested subsequences to complete.
callback	A sequence that is used to handle common tasks, such as serial number inquiry or report logging.
checkbox	A dialog box input 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 exists only in the step type itself. TestStand uses these properties to define how the step type works for all step instances. Step instances do not contain their own copies of class properties.
client sequence file	A sequence file that contains the main sequence a process model invokes to test a UUT. Each client sequence file contains a sequence called `MainSequence`. The process model defines what is constant about your testing process, whereas the client sequence file defines the steps that are unique to the different types of tests you run.
clipboard	A temporary storage area the operating system uses to hold text that is cut, copied, or deleted from a work area.
cluster	A set of ordered, unindexed data elements in LabVIEW of any data type including numeric, Boolean, string, array, or cluster. The elements must be all controls or all indicators.
code module	A program module, such as a Windows Dynamic Link Library (`.dll`) or LabVIEW VI (`.vi`), that contains one or more functions that perform a specific test or other action.
code template	A source file that contains skeleton code. The skeleton code serves as a starting point for the development of code modules for steps that use the step type.
configuration entry point	A sequence in the process model file that configures a feature of the process model. Configuration entry points usually save configuration information in a `.ini` file in the `TestStand\cfg` directory. By default, configuration entry points appear in the **Configure** menu. For example, the default process model contains the configuration entry point: `Config Report Options`. The `Config Report Options` entry point appears as **Report Options** in the **Configure** menu.
connector	Part of a LabVIEW VI or function node that contains its input and output terminals, through which data passes to and from the node.
container property	A property that contains no values, and typically contain multiple subproperties. Container properties are analogous to structures in C/C++ and to clusters in LabVIEW.

context menu	Menus accessed by clicking on an object. Menu options pertain to that object specifically.
control	An input and output device for entering data that appears on a panel or window.
control flow	The sequential order of instructions that determines execution order.
CPU	central processing unit
custom named data type	A data type that you define and name. For example, you might create a `Transmitter` data type that contains subproperties such as `NumChannels` and `PowerLevel`.
custom property	A property that you define in a step type. Each step you create with the step type has its own copy of the custom property. TestStand uses the value that you enter 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

developer	A user profile that usually contains all privileges associated with operating, debugging, and developing sequences and sequence files, but cannot configure user privileges, report options, or database options.
dialog box	A prompt mechanism in which you enter additional information needed to complete a command.
DLL	dynamic link library

E

Edit substep	A substep that the engine calls when editing the step. You invoke the substep with the menu item that appears in the context menu above **Specify Module**. The Edit substep displays a dialog box in which the sequence developer edits the values of custom step properties. For example, the Edit Limits item appears in the context menu for Numeric Limit test steps, and the Edit Pass/Fail Source item appears in the context menu for Pass/Fail test steps.

engine	*See* test executive engine
engine callback	A sequence that TestStand invokes at specific points during execution. You use engine callbacks to tell TestStand to call certain sequences before and after the execution of individual steps, before and after interactive executions, after loading a sequence file, and before unloading a sequence file.
entry points	A sequence in the process model file that TestStand displays as a menu item, such as `Test UUTs`, `Single Pass`, and `Report Options`.
error occurred flag	A Boolean flag, `Step.Result.Error.Occurred`, that indicates whether a run-time error occurred in the step.
execution	An object that contains all the information TestStand needs to run a sequence, its steps, and any subsequences it calls. Typically, the TestStand sequence editor creates a new window for each execution.
execution entry point	A sequence in a process model that runs tests against a UUT. Execution entry points call the `MainSequence` callback in the client sequence file. The default process model contains two execution entry points: `Test UUTs` and `Single Pass`. By default, execution entry points appear in the **Execute** menu. Execution entry points appear in the menu only when the active window contains a sequence file that has a `MainSequence` callback.
execution pointer	A yellow pointer icon that shows the progress of execution by pointing to the currently executing step in the Steps tab.
Execution window	A window in the sequence editor that displays the steps an execution runs. When execution is suspended, the execution window displays the next step to execute and provides single-stepping options. You also can view variables and properties in any active sequence context in the call stack.
expression	A formula that calculates a new value from the values of multiple variable or properties. In expressions, you can access all variables and properties in the sequence context that is active when TestStand evaluates the expression. The following is an example of an expression: `Locals.MidBandFrequency = (Step.HighFrequency + Step.LowFrequency) / 2`

F

front 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.
front-end callback	A common sequence that the sequence editor and run-time operator interfaces call. Front-end callbacks allow multiple applications to share the same implementation for a specific operation. TestStand installs the sequence file `FrontEndCallback.seq`, which contains the front-end callback sequence, `LoginLogout`.
front-end callback sequence file	A sequence file that contains front-end callbacks. TestStand installs the sequence file `FrontEndCallback.seq`, which contains the front-end callback sequence, `LoginLogout`.

G

G	The graphical programming language used to develop LabVIEW applications
global variable	TestStand defines two types of globals: sequence file globals and station globals. Sequence file globals are accessible by any sequence or step in the sequence file. Station globals are accessible by any sequence file loaded on the station. The values of station global variables are persistent across different executions and even across different invocations of TestStand.
GUI	*See* run-time operator interface.

H

hex	hexadecimal
highlight	The way in which input focus appears on a TestStand screen; to move the input focus onto an item.

I

in-process	When executable code runs in the same process space as the client, that is, an ActiveX server in a dynamic-link library(DLL).
instance step type property	A built-in step property that exists in each step instance. Each step that you create with the step type has its own copy of the property. TestStand uses the value you specify for an instance property in the step type as the initial value of the property in each new step that you create. Normally, after you create a step, you can change the values of its instance properties.
interactive mode	When you run steps by selecting one or more steps in a sequence and choosing the **Run Selected Steps** or **Loop Selected Steps** items in the context menu or menu bar. The selected steps in the sequence execute, regardless of any branching logic that the sequence contains. The selected steps run in the order in which they appear in the sequence.

K

kill	To stop a running, terminating, or aborting execution by terminating the thread of the execution without any cleanup of memory. This can leave TestStand in an unreliable state.

L

LabVIEW	Laboratory Virtual Instrument Engineering Workbench. A program development application based on the programming language G and used commonly for test and measurement purposes
list box	A dialog box item that displays a list of possible choices.
local variable	A property of a sequence that holds a value or additional subproperties. Only a step within the sequence can directly access the property value.

M

main sequence	The sequence that initiates the tests on a UUT. The process model invokes the main sequence as part of the overall testing process. The process model defines what is constant about your testing process, whereas main sequences define the steps that are unique to the different types of tests you run.

MB	megabytes of memory
menu bar	Horizontal bar that contains names of main menus.
method	Performs an operation or function on an object.
MFC	Microsoft Foundation Class Library
MHz	megahertz
model callback	A mechanism which allows a sequence file to customize the default behavior of a sequence in the process model.
model sequence file	A special type of sequence file that contains process model sequences. The sequences within the sequence file direct the high-level sequence flow of an execution when testing a UUT.
module adapter	A component that the TestStand engine uses to invoke code in another sequence or in a code module, such as LabVIEW. When invoking code in a code module, the adapter knows how to call it, and how to pass parameters to it.

N

named data type	A type of variable or property that you give a unique name. The data type usually contains multiple subproperties thus creating an arbitrarily complex data structure. All variables or properties that use the data type have the same data structure, but the values they contain can differ.
nested	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.
normal execution	When you start an execution in the sequence editor by selecting the **Run *Sequence Name*** item or one of the process model entry points from the **Execute** menu.
normal sequence file	Any sequence file containing sequences that test UUTs.
numeric property	A 64-bit floating-point value in the IEEE 754 format.

O

object	A service that an ActiveX server makes available to clients.
operator	A user profile that usually contains all privileges associated with operating a test station, but cannot debug sequence executions, edit sequence files, or configure user privileges, station options, report options, and database options.
out-of-process	When executable code does not run in the same process space as the client, such as an ActiveX server in an executable.

P

pop-up menus	*See* context menu
post actions	Actions that TestStand takes depending on the pass/fail status of the step or a custom condition the engine evaluates after executing a step. Post actions allow you to execute callbacks or jump to other steps after executing the step.
Post Step substep	A substep that the engine invokes after calling a step module. A Post Step substep might call a code module that compares the values the step module stored in step properties against limit values that the Edit substep stored in other step properties.
Pre Step substep	A substep that the engine invokes before calling the step module. For example, a Pre Step substep might call a code module that retrieves measurement configuration parameters and stores them into step properties for use by the step module.
preconditions	A set of conditions for a step that must be true for TestStand to execute the step during the normal flow of execution in a sequence.
process model	A series of operations before and after a test executive executes the sequence that performs the tests. Common operations include identifying the UUT, notifying the operator of pass/fail status, generating a test report, and logging results.
property	A container of information, which stores and maintains a setting or attribute of an object. A property can contain a single value, an array of values of the same type, or no value at all. A property can also contain any number of subproperties. Each property has a name.

property-array property	A property containing a value that is an array of subproperties of a single type. In addition to the array of subproperties, property-array properties can contain any number of subproperties of other types.

R

RAM	random-access memory
reference count	Each ActiveX object keeps track of the number of things that reference it. This allows the object to decide when to free the resources it uses.
reference property	*See* ActiveX reference property.
resource string	Text strings stored in an external file so you can alter the strings without directly altering the application.
ROM	read-only memory
root interactive execution	When you run selected steps from a Sequence File window in an independent execution. Root interactive executions do not invoke process models.
run mode	The mode in which you execute a step, such as normal, skip, force pass, force fail.
run-time error	An error condition that forces an execution to terminate. When the error occurs while running a sequence, TestStand jumps to the Cleanup step group, and the error propagates to any calling sequence up through to the top-level sequence.
run-time operator interface	A program that provides a graphical user interface for executing sequences at a production station. Sometimes the sequence editor and run-time operator interfaces are different aspects of the same program.
RTF	rich text format

S

s	seconds
sequence	A series of steps that you specify for execution in a particular order. Whether and when a step is executed can depend on the results of previous steps.

sequence context	A TestStand object that contains references to all global variables and all local variables and step properties in active sequences. The contents of the sequence context changes depending on the currently executing sequence and step.
sequence editor	A program that provides a graphical user interface for creating, editing, and debugging sequences.
sequence file	A file that contains the definition of one or more sequences.
Sequence File window	A separate window within the sequence editor that a sequence file appears in.
sequence globals	Variables you can use to store data relevant to the entire sequence file. Each sequence and step in the sequence file can directly access these globals.
single-valued property	A property that contains a single value. TestStand has four types of these properties: Number properties, String properties, Boolean properties, and ActiveX reference properties.
source code template	A set of source files that contain skeleton code, which serves as a starting point for the development of code modules for steps. TestStand uses the code template when the sequence developer clicks on the **Create Code** button on the Source Code tab in the Specify Module dialog box for a step.
standard named data type	A data type that TestStand defines and names. You can add subproperties to the standard data types, but you cannot delete any of their built-in subproperties. The standard named data types are `Path`, `Error`, and `CommonResults`.
station callback sequence file	A sequence file that contains the station callback sequences. Station callbacks run before and after the engine executes each step in any normal or interactive execution.
station globals	Variables that are persistent across different executions and even across different invocations of the sequence editor or run-time operator interfaces. The TestStand engine maintains the value of station global variables in a file on the run-time computer.
station model	A process model that you select to use for all sequence files for a station. The TestStand installation program establishes `SequentialModel.seq` as the default station model file. You can use the Station Options dialog box to select a different station model.

step	Any action, such calling a test module to perform a specific test, that you can include within a sequence of other actions.
step group	A set of steps in a sequence. A sequence contains the following groups of steps: Setup, Main, and Cleanup. When TestStand executes a sequence, the steps in the Setup group execute first, the steps in the Main group execute next, and the steps in the Cleanup group last.
step module	The code module that a step calls.
step property	A property of a step.
step result	A container property that contains a copy of the subproperties from the `Result` property of a step and additional execution information such as the name of the step and its position in the sequence. TestStand automatically creates a step result as each step executes and places the step result into a result list which TestStand uses to generate its reports.
step status	A string value that indicates the status of a step in an execution. Every step in TestStand has a `Result.Status` property. Although TestStand imposes no restrictions on the values to which the step or its code module can set the status property, TestStand and the built-in step types use and recognize a predefined set of values.
step type	A component that defines a set of custom step properties and standard behavior for each step of that type. All steps of the same type have the same properties, but the values of the properties can differ. Step types define their standard behaviors using substeps.
step-type-specific dialog box	A dialog box that step types display when their Edit substep is invoked. The dialog box lets you modify step properties that are specific to the step type. You invoke the dialog box with the menu item that appears in the context menu above **Specify Module**. For example, the Edit Limits item appears in the context menu for Numeric Limit test steps, and the Edit Pass/Fail Source item appears in the context menu for Pass/Fail test steps.
subsequence	A sequence that another sequence calls. You specify a subsequence call as a step in the calling sequence.
substep	Actions that a step type performs for a step besides calling the step module. You define a substep by selecting an adapter and specifying a module call. TestStand defines three different types of substeps: Edit substep, Pre Step substep, and Post Step substep.

substep module	The code module that a Edit, Pre Step, or Post Step substep calls.
SVGA	Super VGA

T

technician	A user profile that usually contains all privileges associated with operating, and debugging sequences and sequences files, but cannot edit sequence files or configure user privileges, station options, report options, or database options.
template	*See* code template.
terminal	Object or region on a LabVIEW VI node through which data passes.
terminate	To stop an execution by halting the normal execution flow, and running all the Cleanup step groups in the sequences on the call stack.
test executive engine	A module or set of modules that provide an API for creating, editing, executing, and debugging sequences. A sequence editor or run-time execution operator interface uses the services of a test executive engine.
test module	A code module that performs a test.

U

Unit Under Test (UUT)	The device or component that you are testing.
user manager	The component of the TestStand engine that maintains a list of users, their user names and passwords, and their privileges. You can access the user manager from the User Manager window in the sequence editor.

V

variables	Properties that you can freely create in certain contexts. You can have variables that are global to a sequence file or local to a particular sequence. You can also have station global variables.
variables window	A window that shows the values of all the currently active variables or properties.

VI	Virtual instrument.
VI library	Special file of type `.LLB` that contains a collection of related VIs for a specific use.

W

Watch window	A window that shows the values of user-selectable variables and expressions that are currently active.
window	A working area that supports specific tasks related to developing and executing programs.
wire	Tool used to define data paths between source and sink terminals.

Index

A

ActiveX, using in code modules, 10-1 to 10-22
- LabVIEW test virtual instruments, 10-2 to 10-14
  - Array Bounds dialog box (figure), 10-3
  - creating sequence and virtual instrument tests, 10-2 to 10-12
  - Display TestStandWaveform.vi block diagram (figure), 10-12
  - Display TestStandWaveform.vi front panel (figure), 10-11
  - Generate TestStandWaveform.vi block diagram (figure), 10-10
  - GenerateWaveform.vi block diagram (figure), 10-8
  - GenerateWaveform.vi control panel (figure), 10-7
  - GenerateWaveform.vi front panel (figure), 10-5
  - Insert Locals Array of Numeric (figure), 10-3
  - Locals.Arraydata variable (figure), 10-13
  - running the sequence, 10-12 to 10-14
  - setting up for example, 10-2
  - TestStand control palette (figure), 10-6
  - TestStand function palette (figure), 10-9
- LabWindows/CVI code modules, 10-14 to 10-22
  - Array Bounds dialog box (figure), 10-15
  - creating sequence and tests, 10-14 to 10-20
  - Generated GenerateTestStandWaveform source (figure), 10-17
  - Insert Locals Array of Numeric (figure), 10-15
  - Locals.Arraydata values (figure), 10-22
  - running the sequence, 10-21
  - setting up for example, 10-14

ActiveX Automation Adapter, 1-7

Adapter Configuration dialog box
- debugging LabVIEW VIs (figure), 6-2
- debugging LabWindows/CVI DLL, 6-24

architecture of TestStand (figure), 1-5

Array Bounds dialog box
- using LabVIEW test virtual instruments with ActiveX (figure), 10-3
- using LabWindows/CVI code modules with ActiveX (figure), 10-15

B

Batch process model
- overview, 1-9
- running sequences, 2-12 to 2-15

Batch Synchronization control, Synchronization tab, 3-15

Breakpoint control, Run Options tab, 3-10

C

callbacks, 8-1 to 8-7
- adding callbacks to a sequence (figure), 8-6
- customizing report using callbacks, 12-21 to 12-24
  - Callbacks dialog box (figure), 12-21
  - completed ModifyReportHeader sequence (figure), 12-23
  - inserting string local in ModifyReportHeader (figure), 12-22

ModifyReportHeader callback sequence (figure), 12-21
new HMTL header (figure), 12-24
overriding process model callback, 8-1 to 8-7
setting up for example, 8-1
Test UUTs Sequence (figure), 8-4
TestStand Sequential Model callbacks (figure), 8-3
calling sequences dynamically and passing parameters, 11-6 to 11-13
adding step to sequence, 11-6 to 11-10
dynamically calling a sequence (figure), 11-10
Edit Sequence Call dialog box (figure), 11-9
INTELProcessor.seq in Call Stack pane (figure), 11-11
running a sequence, 11-10 to 11-13
sequence parameters in Context tab (figure), 11-12
calling subsequences, 3-16 to 3-18
C/CVI Standard Prototype Adapter
creating step module, 12-13 to 12-16
debugging LabWindows/CVI DLL, 6-23 to 6-37
Adapter Configuration dialog box, 6-24
C/CVI Standard Adapter Configuration (figure), 6-25
creating C/CVI code module test, 6-24 to 6-35
debug procedure, 6-35 to 6-37
Edit CVI Module Call dialog box
Module tab (figure), 6-26
Source Code tab (figure), 6-27
Edit Numeric Limits Test dialog box (figure), 6-33
generated results from Create Code command (figure), 6-28
Numeric Format dialog box (figure), 6-34
setting up for example, 6-24
stepping into GetFrequency function (figure), 6-36
tErrorData parameter, GetFrequency prototype function, 6-29 to 6-30
tTestData parameter, GetFrequency prototype function, 6-29
overview, 1-7
selecting for new step, 3-2
Choose Code Template dialog box (figure), 6-41
code modules
creating
LabVIEW DLL function, 6-16 to 6-18
LabVIEW VIs, 6-2 to 6-10
LabWindows/CVI DLL
using C/CVI Standard Prototype Adapter, 6-24 to 6-35
using DLL Flexible Prototype Adapter, 6-37 to 6-43
using ActiveX, 10-1 to 10-22
LabVIEW test virtual instruments, 10-2 to 10-14
LabWindows/CVI code modules, 10-14 to 10-22
Configure Message Box Step dialog box (figure), 4-3
Context tab, Execution window, 5-8 to 5-10
conventions used in manual, *iv*
converting Test Executive sequences
LabVIEW Test Executive sequences, 13-1
LabWindows/CVI Test Executive sequences, 13-2
customer education, A-1
customizing reports. *See* report customization.

D

debugging
- LabVIEW DLL function, 6-12 to 6-23
  - adding parameter expressions, 6-19 to 6-20
  - building DLL code module, 6-16 to 6-18
  - calling the DLL function, 6-18 to 6-22
  - Clock Frequency Function diagram (figure), 6-15
  - Clock Frequency Function front panel (figure), 6-14
  - creating virtual instrument code, 6-13 to 6-16
  - debugging procedure, 6-23
  - Define VI Prototype dialog box (figure), 6-17
  - Edit DLL Call dialog box (figure), 6-19
  - Edit Numeric Limits Test dialog box, 6-21
  - VI Properties dialog box (figure), 6-16
- LabVIEW VIs, 6-1 to 6-12
  - Adapter Configuration dialog box, 6-2
  - adding controls to front panel, 6-7
  - Clock Frequency.vi block diagram (figure), 6-8
  - Clock Frequency.vi front panel (figure), 6-7
  - creating virtual instrument code module, 6-2 to 6-10
  - debug procedure for virtual instrument code module, 6-11 to 6-12
  - Edit Numeric Limits Test dialog box (figure), 6-9
  - LabVIEW Adapter Configuration dialog box (figure), 6-3
  - LabVIEW Step Module information (figure), 6-4
  - New Clock Frequency VI (figure), 6-5
  - Numeric Format dialog box (figure), 6-10
  - saving the VI, 6-76-8
  - setting up for example, 6-2
- LabWindows/CVI DLL
  - using C/CVI Standard Prototype Adapter, 6-23 to 6-37
    - Adapter Configuration dialog box, 6-24
    - C/CVI Standard Adapter Configuration (figure), 6-25
    - creating C/CVI code module test, 6-24 to 6-35
    - debug procedure, 6-35 to 6-37
    - Edit CVI Module Call—Module tab dialog box (figure), 6-26
    - Edit CVI Module Call—Source Code tab dialog box (figure), 6-27
    - Edit Numeric Limits Test dialog box (figure), 6-33
    - generated results from Create Code command (figure), 6-28
    - Numeric Format dialog box (figure), 6-34
    - setting up for example, 6-24
    - stepping into GetFrequency function (figure), 6-36
    - tErrorData parameter, GetFrequency prototype function, 6-29 to 6-30
    - tTestData parameter, GetFrequency prototype function, 6-29
  - using DLL Flexible Prototype Adapter, 6-37 to 6-50
    - building DLL, 6-44 to 6-47

Choose Code Template dialog box (figure), 6-41
creating C/CVI code module, 6-37 to 6-43
debug procedure, 6-48 to 6-50
Edit DLL dialog box for LabWindows/CVI Code Module, 6-38
Edit Numeric Limits Test dialog box (figure), 6-43
Numeric Format dialog box (figure), 6-44
parameter control table of values, 6-39 to 6-40
Prototypes Conflict dialog box, 6-42
setting up for example, 6-37
sequences, 4-1 to 4-7
Configure Message Box Step dialog box (figure), 4-3
paused execution, 4-3 to 4-4
run-time interface, 7-4 to 7-5
single-stepping toolbar buttons, 4-5
step mode execution, 4-1 to 4-7
Steps view for Execution window (figure), 4-6
Define VI Prototype dialog box (figure), 6-17
development workspace of sequence editor, 2-3
directly running sequences, 2-9 to 2-10
DLL Flexible Prototype Adapter
creating step module, 12-16 to 12-20
debugging LabVIEW DLL function, 6-12 to 6-23
adding parameter expressions, 6-19 to 6-20
building DLL code module, 6-16 to 6-18
calling the DLL function, 6-18 to 6-22
Clock Frequency Function diagram (figure), 6-15
Clock Frequency Function front panel (figure), 6-14
creating virtual instrument code, 6-13 to 6-16
debugging procedure, 6-23
Define VI Prototype dialog box (figure), 6-17
Edit DLL Call dialog box (figure), 6-19
Edit Numeric Limits Test dialog box, 6-21
VI Properties dialog box (figure), 6-16
debugging LabWindows/CVI DLL, 6-37 to 6-50
building DLL, 6-44 to 6-47
Choose Code Template dialog box (figure), 6-41
creating C/CVI code module, 6-37 to 6-43
debug procedure, 6-48 to 6-50
Edit DLL dialog box for LabWindows/CVI Code Module, 6-38
Edit Numeric Limits Test dialog box (figure), 6-43
Numeric Format dialog box (figure), 6-44
parameter control table of values, 6-39 to 6-40
Prototypes Conflict dialog box, 6-42
setting up for example, 6-37
overview, 1-7

E

Edit C/CVI Module Call dialog box (figure), 3-5
Edit DLL Call dialog box (figure), 6-19
Edit DLL dialog box for LabWindows/CVI Code Module, 6-38

Edit Numeric Limits Test dialog box
  debugging LabVIEW DLL function (figure), 6-21
  debugging LabVIEW VI (figure), 6-9
  debugging LabWindows/CVI DLL (figure), 6-33
Edit Sequence Call dialog box (figure), 3-17
Edit Statement Step dialog box (figure), 5-3
editing steps in a sequence, 3-1 to 3-18
  adding new steps, 3-1 to 3-3
  selecting module adapter, 3-2
  calling subsequence from sequence, 3-16 to 3-18
  changing priorities, 3-5 to 3-16
  predefined step types, 3-1 to 3-2
  setting up for example, 3-1
  specifying test module, 3-4 to 3-5
entry points, 1-8
execution, interactive. *See* interactive execution.
execution pointer, 2-10
Execution tab, Station Options dialog box (figure), 2-8
Execution window
  Context tab, 5-8 to 5-10
  running sequence directly, 2-9
  Watch Expression pane, 5-11 to 5-12
Expression Browser dialog box (figure), 5-4
Expressions tab, Step Properties dialog box, 3-15 to 3-16
external report viewers, 12-6 to 12-7

H

HTBasic Adapter, 1-7

I

Ignore Tun-time Errors control, Run Options tab, 3-10
Insert Local Context Menu command (figure), 5-2
installation of TestStand, 1-1 to 1-3
  files installed by setup program (table), 1-2 to 1-3
  mass compiling VIs from older versions (note), 1-2
  minimum system requirements, 1-1
  procedural steps, 1-1 to 1-2
interactive execution, 11-1 to 11-5
  breakpoint during (figure), 11-3
  looping on selected steps during execution (figure), 11-4
  running steps as separate execution, 11-1 to 11-3
  running steps during execution, 11-4 to 11-5
  selecting multiple steps in Sequence file window (figure), 11-2

L

LabVIEW Adapter Configuration dialog box (figure), 6-3
LabVIEW DLL function, debugging, 6-12 to 6-23
  adding parameter expressions, 6-19 to 6-20
  building DLL code module, 6-16 to 6-18
  calling the DLL function, 6-18 to 6-22
  Clock Frequency Function diagram (figure), 6-15
  Clock Frequency Function front panel (figure), 6-14
  creating virtual instrument code, 6-13 to 6-16
  debugging procedure, 6-23
  Define VI Prototype dialog box (figure), 6-17
  Edit DLL Call dialog box (figure), 6-19

Edit Numeric Limits Test dialog box, 6-21
VI Properties dialog box (figure), 6-16
LabVIEW Standard Prototype Adapter
creating step module, 12-10 to 12-12
debugging LabVIEW VIs, 6-1 to 6-12
overview, 1-7
LabVIEW Test Executive sequences, converting, 13-1
LabVIEW test virtual instruments, using with ActiveX, 10-2 to 10-14
Array Bounds dialog box (figure), 10-3
creating sequence and virtual instrument tests, 10-2 to 10-12
Display TestStandWaveform.vi block diagram (figure), 10-12
Display TestStandWaveform.vi front panel (figure), 10-11
Generate TestStandWaveform.vi block diagram (figure), 10-10
GenerateWaveform.vi block diagram (figure), 10-8
GenerateWaveform.vi control panel (figure), 10-7
GenerateWaveform.vi front panel (figure), 10-5
Insert Locals Array of Numeric (figure), 10-3
Locals.Arraydata variable (figure), 10-13
running the sequence, 10-12 to 10-14
setting up for example, 10-2
TestStand control palette (figure), 10-6
TestStand function palette (figure), 10-9
LabVIEW VIs, debugging, 6-1 to 6-12
Adapter Configuration dialog box, 6-2
adding controls to front panel, 6-7
Clock Frequency.vi block diagram (figure), 6-8
Clock Frequency.vi front panel (figure), 6-7
creating virtual instrument code module, 6-2 to 6-10
debug procedure for virtual instrument code module, 6-11 to 6-12
Edit Numeric Limits Test dialog box (figure), 6-9
LabVIEW Adapter Configuration dialog box (figure), 6-3
LabVIEW Step Module information (figure), 6-4
New Clock Frequency VI (figure), 6-5
Numeric Format dialog box (figure), 6-10
saving the VI, 6-76-8
setting up for example, 6-2
LabWindows/CVI code modules, using with ActiveX, 10-14 to 10-22
Array Bounds dialog box (figure), 10-15
creating sequence and tests, 10-14 to 10-20
Generated GenerateTestStandWaveform source (figure), 10-17
Insert Locals Array of Numeric (figure), 10-15
Locals.Arraydata values (figure), 10-22
running the sequence, 10-21
setting up for example, 10-14
LabWindows/CVI DLL, debugging
using C/CVI Standard Prototype Adapter, 6-23 to 6-37
Adapter Configuration dialog box, 6-24
C/CVI Standard Adapter Configuration (figure), 6-25
creating C/CVI code module test, 6-24 to 6-35
debug procedure, 6-35 to 6-37
Edit CVI Module Call dialog box
Module tab (figure), 6-26
Source Code tab (figure), 6-27
Edit Numeric Limits Test dialog box (figure), 6-33

generated results from Create Code command (figure), 6-28
Numeric Format dialog box (figure), 6-34
setting up for example, 6-24
stepping into GetFrequency function (figure), 6-36
tErrorData parameter, GetFrequency prototype function, 6-29 to 6-30
tTestData parameter, GetFrequency prototype function, 6-29
using DLL Flexible Prototype Adapter, 6-37 to 6-50
building DLL, 6-44 to 6-47
Choose Code Template dialog box (figure), 6-41
creating C/CVI code module, 6-37 to 6-43
debug procedure, 6-48 to 6-50
Edit DLL dialog box for LabWindows/CVI Code Module, 6-38
Edit Numeric Limits Test dialog box (figure), 6-43
Numeric Format dialog box (figure), 6-44
parameter control table of values, 6-39 to 6-40
Prototypes Conflict dialog box, 6-42
setting up for example, 6-37
LabWindows/CVI Test Executive sequences, converting, 13-2
learning to use TestStand, 1-3 to 1-4
Load option, Run Options tab, 3-8 to 3-9
loading sequence files, 2-4 to 2-6
Open dialog box (figure), 2-4
run-time interface, 7-1 to 7-4
sample sequence file window (figure), 2-5
View ring for selecting files (figure), 2-6
viewing tab contents, 2-6
local variables. *See* variables.
Lock Name or Reference Expression control, Synchronization tab, 3-14 to 3-15
Login dialog box (figure), 2-1
Loop Options tab, Step Properties dialog box, 3-12 to 3-13
illustration, 3-12
values for Loop Type, 3-12 to 3-13

M

menu bar of sequence editor, 2-2
Model Options dialog box (figure), 2-13
module adapters
overview, 1-7
specifying for new step, 3-2
types of adapters, 1-7 to 1-8

N

New User dialog box, 9-4
NI Developer Zone, A-1
Numeric Format dialog box
debugging LabVIEW DLL function (figure), 6-22
debugging LabVIEW VIs (figure), 6-10
debugging LabWindows/CVI DLL (figure), 6-34

O

Open dialog box (figure), 2-4

P

Parallel process model, 1-9
Pass/Fail Test, inserting in new step, 3-3
Post Actions tab, Step Properties dialog box, 3-11
Precondition Evaluation in Interactive Mode control, Run Options tab, 3-9

Preconditions dialog box
changing step priorities (figure), 3-7 to 3-8
creating variables (figure), 5-7
privileges, granting. *See* User Manager.
process models
Batch process model
overview, 1-9
running sequences, 2-12 to 2-15
changing, 1-9
default TestStand process model, 1-8 to 1-9
definition, 1-9
overriding process model callback, 8-1 to 8-7
Parallel model, 1-9
Sequential process model, 2-11 to 2-12
properties
changing step properties, 3-5 to 3-16
determined by type of step, 10-1
first-level properties of sequence context (table), 5-10
Properties command, 3-6
Prototypes Conflict dialog box, 6-42

R

Record Results control, Run Options tab, 3-10
report customization, 12-1 to 12-24
adding new step properties to report, 12-7 to 12-20
creating step module, 12-10 to 12-20
using C/CVI Standard Prototype Adapter, 12-13 to 12-16
using DLL Flexible Prototype Adapter, 12-16 to 12-20
using LabVIEW Standard Prototype Adapter, 12-10 to 12-12
creating step type, 12-7 to 12-10
Insert Numeric Array Context menu (figure), 12-8
Numeric Array Properties Context menu (figure), 12-9
setting up for example, 12-7
callbacks for adding items to report, 12-21 to 12-24
configuring test report options, 12-1 to 12-5
test report in HTML format (figure), 12-5
test report in text format (figure), 12-4
UUT report setting (figure), 12-2
external report viewers, 12-6 to 12-7
setting up for example, 12-1
requirements for installing TestStand, 1-1
Run Mode control, Run Options tab, 3-9
Run Options tab, Step Properties dialog box, 3-8 to 3-9
Breakpoint control, 3-10
Ignore Tun-time Errors control, 3-10
illustration, 3-8
Load option, 3-8 to 3-9
Precondition Evaluation in Interactive Mode control, 3-9
Record Results control, 3-10
Run Mode control, 3-9
Step Failure Causes Sequence Failure control, 3-10
TestStand Window Activation control, 3-10
Unload option, 3-9
running sequences, 2-7 to 2-15
directly, 2-9 to 2-10
multiple executions, in run-time interface, 7-6
run-time interface, 7-4 to 7-5
setting up tracing options, 2-7 to 2-8

using Batch process model, 2-12 to 2-15
using Sequential process model, 2-11 to 2-12
run-time operator interfaces, 7-1 to 7-6
loading sequences, 7-1 to 7-4
LabWindows/CVI operator interface (figure), 7-2
Open Sequence (figure), 7-3
overview, 1-6
running and debugging sequences, 7-4 to 7-5
running multiple executions, 7-6

S

Sequence Adapter, 1-7
sequence context
definition, 10-1
first-level properties (table), 5-10
sequence editor
development workspace, 2-3
main window (figure), 2-2
menu bar, 2-2
overview, 1-6
status bar, 2-3 to 2-4
toolbar, 2-3
sequence file global variables, 5-1
sequences
calling dynamically and passing parameters, 11-6 to 11-13
adding step to sequence, 11-6 to 11-10
dynamically calling a sequence (figure), 11-10
Edit Sequence Call dialog box (figure), 11-9
INTELProcessor.seq in Call Stack pane (figure), 11-11
running a sequence, 11-10 to 11-13
sequence parameters in Context tab (figure), 11-12
debugging, 4-1 to 4-7
Configure Message Box Step dialog box (figure), 4-3
paused execution, 4-3 to 4-4
run-time interface, 7-4 to 7-5
single-stepping toolbar buttons, 4-5
step mode execution, 4-1 to 4-7
Steps view for Execution window (figure), 4-6
editing steps in a sequence, 3-1 to 3-18
adding new steps, 3-1 to 3-3
selecting module adapter, 3-2
calling subsequence from sequence, 3-16 to 3-18
changing priorities, 3-5 to 3-16
predefined step types, 3-1 to 3-2
setting up for example, 3-1
specifying test module, 3-4 to 3-5
loading sequence files, 2-4 to 2-6
Open dialog box (figure), 2-4
run-time interface, 7-1 to 7-4
sample sequence file window (figure), 2-5
View ring for selecting files (figure), 2-6
viewing tab contents, 2-6
overview, 2-7
running, 2-7 to 2-15
directly, 2-9 to 2-10
multiple executions, in run-time interface, 7-6
run-time interface, 7-4 to 7-5
setting up tracing options, 2-7 to 2-8
using Batch process model, 2-12 to 2-15
using Sequential process model, 2-11 to 2-12
Sequential process model for running sequences, 2-11 to 2-12
Specify Module dialog box, 3-4 to 3-5
starting TestStand, 2-1 to 2-2

station global variables, 5-1, 10-1
status bar of sequence editor, 2-3 to 2-4
Step Failure Causes Sequence Failure control
Run Options tab, 3-10
Step Into command, Debug menu, 4-4
step mode execution, 4-1 to 4-7
  Configure Message Box Step dialog box (figure), 4-3
  paused execution, 4-3 to 4-4
  single-stepping toolbar buttons, 4-5
  Steps view for Execution window (figure), 4-6
Step Out command, Debug menu, 4-4
Step Over command, Debug menu, 4-4
step properties
  adding to reports. *See* report customization.
  changing, 3-5 to 3-16
Step Properties dialog box, 3-5 to 3-16
  Expressions tab, 3-15 to 3-16
  illustration, 3-6
  Loop Options tab, 3-12 to 3-13
  Post Actions tab, 3-11
  Run Options tab, 3-8 to 3-9
  Synchronization tab, 3-14 to 3-15
step type, creating, 12-7 to 12-10
steps, 3-1 to 3-18
  adding new steps, 3-1 to 3-3
    selecting module adapter, 3-2
  calling subsequence from sequence, 3-16 to 3-18
  changing properties, 3-5 to 3-16
  predefined step types, 3-1 to 3-2
  setting up for example, 3-1
  specifying test module, 3-4 to 3-5
subsequences, calling, 3-16 to 3-18
Synchronization tab, Step Properties dialog box, 3-14 to 3-15
  Batch Synchronization control, 3-15
  illustration, 3-14
  Lock Name or Reference Expression control, 3-14 to 3-15
  Use Lock to allow Only One Thread to Execute the Step control, 3-14
system integration, by National Instruments, A-1

T

technical support resources, A-1 to A-2
Test Executive Engine, 1-7
Test Executive sequences, converting
  LabVIEW Test Executive sequences, 13-1
  LabWindows/CVI Test Executive sequences, 13-2
test module, specifying, 3-4 to 3-5
Test Simulator dialog box (figure), 2-10
TestStand
  installation, 1-1 to 1-3
  learning to use, 1-3 to 1-4
  major software components, 1-5 to 1-9
    module adapters, 1-7 to 1-8
    process models, 1-8 to 1-9
    run-time operator interfaces, 1-6
    sequence editor, 1-6
    Test Executive Engine, 1-7
  overview, 1-4
  starting, 2-1 to 2-2
  system architecture of (figure), 1-5
TestStand Window Activation control, 3-10
toolbar of sequence editor, 2-3
tracing options, setting up, 2-7 to 2-8

U

Unload option, Run Options tab, 3-9
Use Lock to allow Only One Thread to Execute the Step control, Synchronization tab, 3-14
User Manager, 9-1 to 9-6

adding new users, 9-3 to 9-6
creating new profile, 9-4 to 9-6
granting privileges, 9-2
New User dialog box, 9-4
setting up for example, 9-1
User Manager window (figure), 9-3
viewing current users, 9-1 to 9-2

V

variables
creating and using local variables, 5-1 to 5-8
Edit Statement Step dialog box (figure), 5-3
Expression Browser dialog box (figure), 5-4
Insert Local Context Menu command (figure), 5-2
Preconditions for the Loop End Step dialog box, 5-7
setting up for example, 5-4
local variables, 5-1
purpose and use, 10-1
sequence file global variables, 5-1
station global variables, 5-1, 10-1
using Context tab, 5-8 to 5-10
using Watch Expression pane, 5-11 to 5-12
VI Properties dialog box (figure), 6-16
VIs (virtual instruments)
creating virtual instrument code module
debugging LabVIEW DLL function, 6-13 to 6-16
debugging LabVIEW VIs, 6-2 to 6-10
mass compiling VIs from older versions (note), 1-2
using ActiveX in LabVIEW test virtual instruments, 10-2 to 10-14

W

Watch Expression pane, Execution window, 5-11 to 5-12
Web support from National Instruments, A-1
Worldwide technical support, A-2