Contents

HP TestExec SL Getting Started Book

E2011-90017 — Software Rev. 3.00 — Rev. E - January, 199

3. Concepts

Notice

Warranty Information

Restricted Rights Legend

Printing History

E1074-90000 — Software Rev. 1.00 — First printing - August, 1995

E1074-90005 — Software Rev. 1.50 — Rev. A - March, 1996

Note The documentation expanded into a multi-volume set of books at Rev. B.

E1074-90006 — Software Rev. 1.51 — Rev. B - June, 1996

E2011-90010 — Software Rev. 2.00 — Rev. C - January, 1997

E2011-90013 — Software Rev. 2.10 — Rev. D - May, 1997

E2011-90017 — Software Rev. 3.00 — Rev. E - January, 1998

About This Manual

This manual provides beginners with an introduction to HP TestExec SL. It describes the product at an overview level, tells how to install and run the HP TestExec SL software, and introduces concepts used in HP TestExec SL and further described in the other users' manuals.

Conventions Used in this Manual

Vertical bars denote a hierarchy of menus and commands, such as:

View | Listing | Actions

Here, you are being told to choose the Actions command that appears when you expand the Listing command in the View menu.

If a form uses tabs to organize its contents, the name of a tab may appear in the hierarchy of menus and commands. For example, the Options dialog box has a tab named Search Paths. A reference to that tab looks like this:

View | Options | Search Paths

To make the names of functions stand out in text yet be concise, the names typically are followed by "empty" parentheses—i.e., `MyFunction()`—that do not show the function's parameters.

Some programming examples use the C++ convention for comments, which is to begin commented lines with two slash characters, like this:

```
// This is a comment
```

C++ compilers also will accept the C convention of:

```
/* This is a comment */
```

The C++ convention is used here simply because it results in shorter line lengths, which make examples fit better on a printed page. If you are using a C-only compiler, be sure to follow the C convention.

1

Introduction to HP TestExec SL

This chapter describes HP TestExec SL’s features at a high level and broadly describes system integration, which is the process of combining hardware and software to create a test system.

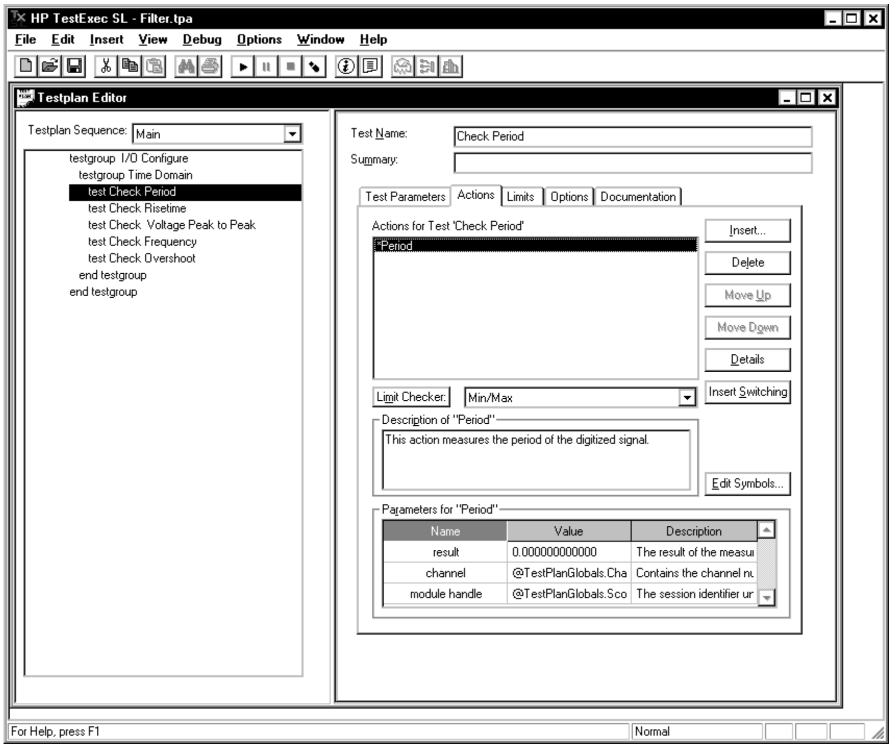
What is HP TestExec SL?

HP TestExec SL is a test executive designed for high-volume, high-throughput functional test applications. Its test development and execution environment provides you with a choice of programming languages, instrumentation, fast and flexible switch management, and a customizable operator interface. It also includes library features that promote the maintenance and reuse of code.

HP TestExec SL contains three interrelated tools in a single environment:

- Test Executive

  The Test Executive is used to develop tests and assemble them into testplans. It also provides features for running and debugging testplans. The Test Executive can have multiple personalities, which means its appearance can change to suit the kind of task being done. For example, it presents one appearance when used as a test development environment, and another when used by production operators. Its personality is determined by which login you use. The test development personality shown below is used when developing tests.



Testplan Sequence: Main

```
testgroup  I/O Configure
  testgroup Time Domain
    test Check Period
    test Check Risetime
    test Check  Voltage Peak to Peak
    test Check Frequency
    test Check Overshoot
  end testgroup
end testgroup
```

Test Name: Check Period
Summary:

Actions for Test 'Check Period'
*Period

Limit Checker: Min/Max

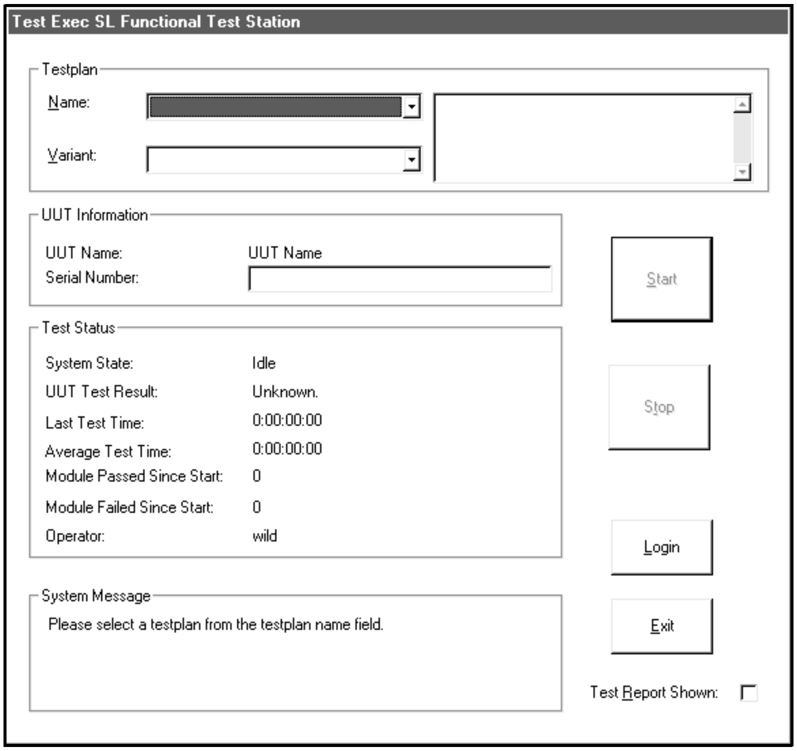
Description of "Period"
This action measures the period of the digitized signal.

Parameters for "Period"

Name	Value	Description
result	0.000000000000	The result of the measu
channel	@TestPlanGlobals.Cha	Contains the channel nu
module handle	@TestPlanGlobals.Sco	The session identifier ur

For Help, press F1 | Normal

An alternative, operator personality is customizable to meet the needs of a specific production environment. Its default implementation, which provides the basic control features and status information needed in a typical production environment, is shown below.



UUT Name: UUT Name
Serial Number:

Test Status

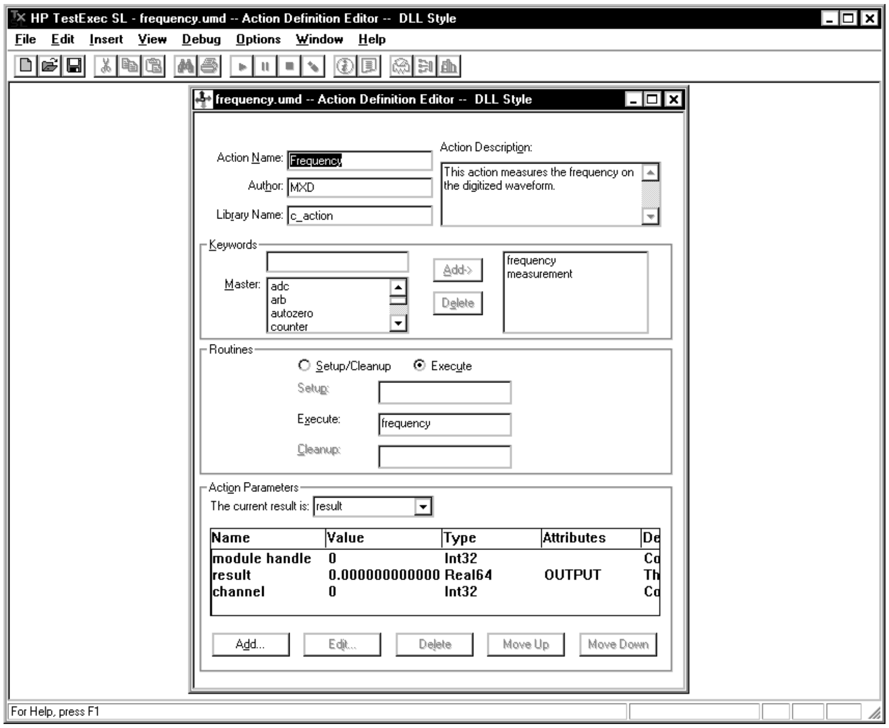
System State: Idle
UUT Test Result: Unknown.
Last Test Time: 0:00:00:00
Average Test Time: 0:00:00:00
Module Passed Since Start: 0
Module Failed Since Start: 0
Operator: wild

System Message

Please select a testplan from the testplan name field.

- Action Definition Editor

  The Action Definition Editor is used to create actions, which are the building blocks from which tests are created. It lets you add code written in the programming language of your choice into the framework of HP TestExec SL.[1]


HP TestExec SL - frequency.umd -- Action Definition Editor -- DLL Style
File Edit Insert View Debug Options Window Help
frequency.umd -- Action Definition Editor -- DLL Style
Add->
Delete
Add...
Edit...
Delete
Move Up
Move Down
For Help, press F1

Action Name: Frequency
Author: MXD
Library Name: c_action
Action Description: This action measures the frequency on the digitized waveform.

Keywords
Master: adc, arb, autozero, counter
frequency
measurement

Routines
Setup/Cleanup [✓ Execute]
Setup:
Execute: frequency
Cleanup:

Action Parameters
The current result is: result

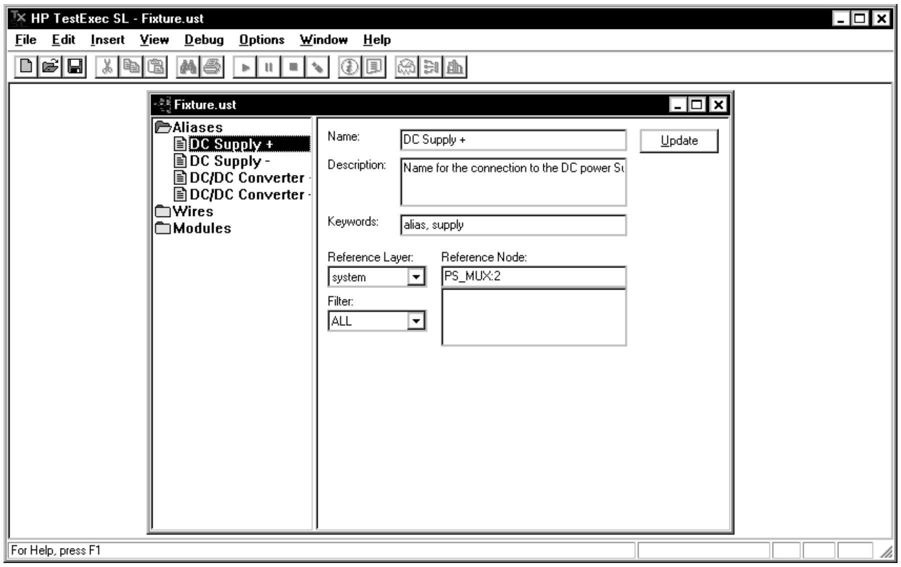
Name	Value	Type	Attributes	De
module handle	0	Int32		Co
result	0.000000000000	Real64	OUTPUT	Th
channel	0	Int32		Co

- Switching Topology Editor

  The Switching Topology Editor is used to define switchable connections (low-level hardware) and the wiring inside a fixture used with the Test

1. C/C++, HP BASIC for Windows, HP VEE, and National Instruments LabVIEW are supported.

Executive. Also, it is used to make the Test Executive aware of hardware modules that are available as resources during testing.



Name: DC Supply +
Description: Name for the connection to the DC power Su
Keywords: alias, supply
Reference Layer: system
Reference Node: PS_MUX:2
Filter: ALL

What Makes HP TestExec SL Different?

Test executives have existed for as long as there has been a need to do repeatable testing in a production environment. Traditionally, test executives have been custom, “roll your own” programs whose features included sequencing, testing, checking pass/fail limits, error handling, displaying status information to users, and more. Tests were written in the same programming language as the rest of the test executive. To keep the main program from growing too large and unwieldy, tests usually were invoked by calls to the functions or subroutines in which the tests resided.

One thing that makes HP TestExec SL unique in the world of test executives is that it more clearly distinguishes between sequencing operations and testing operations. Sequencing—i.e., executing tests in a repeatable order—is handled via a graphical user interface that executes tests written in any of several standard programming languages. Unlike typical test executives, HP TestExec SL’s sequencer has no programming language associated with it. Instead, you use menu commands or icons to specify the sequence in which tests execute, set up global variables used to pass values between tests, and other features of testing.

In HP TestExec SL, tests are a highly structured concept. They combine user-written measurement code with system-supplied limits checking, datalogging, test report generation, and pre- and post-conditioning features. Because the language in which you write tests is not dictated by the language in which the test executive is implemented, you can write them in any of several languages.

From the viewpoint of the sequencer, tests themselves are “atomic” insofar as they cannot be divided into smaller units unless you write the code in them that way. For example, you could conceivably use the sequencer to execute one huge test that made multiple measurements and did everything. But tests usually are small, and acquire and check only a single operating parameter of the unit under test. Besides making tests more manageable, this improves their reusability.

Another key difference of HP TestExec SL is that it is based on an underlying technology from Hewlett-Packard called HP TestCore, which is an open, standardized framework for creating or modifying test systems.

Using this framework lets Hewlett-Packard enhance the functionality and performance of HP TestExec SL while maintaining a high degree of compatibility with existing tests. This improves the long-term usefulness of the code you write today.

Flexibility is another major benefit that HP TestCore brings to HP TestExec SL. If desired, your tests can make calls to numerous API (application programming interface) functions that manipulate the hardware and data used by your test system. For example, the easiest way to control switching hardware is through graphical features built into HP TestExec SL. But if you need even greater control over switching, such as when changing switching paths “on the fly” in the middle of a test, you can call an API that provides lower-level access to switching hardware. In a similar fashion, you can control many other aspects of HP TestExec SL’s operation at both high and low levels.

The Benefits of Reusable Code

Why Code Is Not Reused

Although developing the programming code in measurement routines—i.e., code that makes a measurement—can be time consuming, all too often the finished routines must be regarded as unique rather than reusable. The lack of standardized tools and methods makes it difficult to keep track of which routines are available and their characteristics, such as what they do and who wrote them. Another traditional problem with measurement routines is that their form may differ substantially across the various programming and application platforms. For example, a routine written in BASIC may look very different from a routine written in ANSI C. This tendency to make each measurement routine a custom application also limits its portability from one environment to another.

In a similar fashion, test procedures—i.e., groups of measurement routines that comprise a test—may not be reused. Prior to HP TestExec SL, test developers typically wrote a single “test procedure” that contained all the code required to do the tasks needed to measure a performance characteristic of a specific unit. A group of test procedures executed in sequence became a “test sequence” or “testplan.” Often, these test procedures and testplans did the same kinds of tasks; for example, each might set up the instruments or conditions needed for the measurement, make the measurement, evaluate the results, and such. Although these procedures fundamentally tended to be more alike than different, they seemed like poor candidates for reuse. For example, their lack of modularity made them difficult to modify or maintain. Also, there was no provision for managing them, which meant that other potential users were unaware of their characteristics or, in many cases, of their existence.

How HP TestExec SL Encourages Reusability

Continuing with the example of measurement routines and test procedures, suppose that test developers had access to a well-maintained library of measurement routines written in a standardized, reusable format that was compatible across platforms. These routines might include those developed in-house or prewritten routines purchased from other vendors. Then a test developer might create a testplan simply by choosing from a catalog of existing routines and assigning a sequence to them. Because it reduces the ongoing necessity to “reinvent the wheel,” this convenient reuse of routines—i.e., reduction in programming effort—is a major benefit of HP TestExec SL.

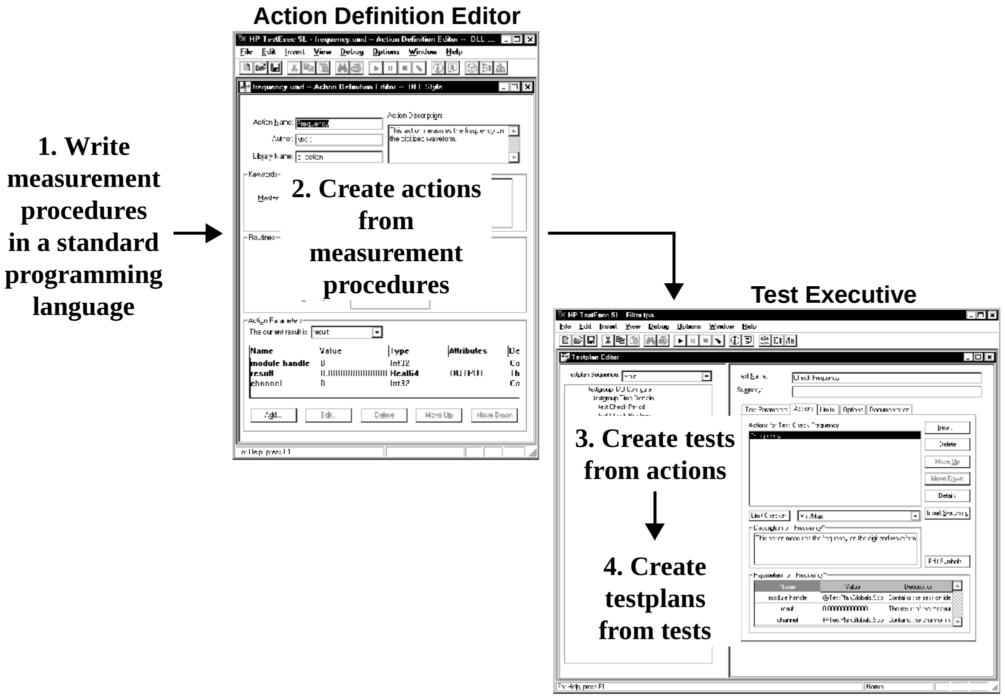
When existing measurement routines cannot meet the needs of the test developer, HP TestExec SL's architecture provides standard methods for modifying or enhancing existing measurement routines. Once these new routines have been developed and debugged, they too become candidates for reuse. Unlike traditional test routines, which may all be incorporated into a single, large program, HP TestExec SL uses a modular approach that is easy to modify and maintain.

Another benefit of HP TestExec SL is that it provides tools for administering the collection—i.e., “library”—of measurement routines. For example, it is easy to search for existing routines or procedures that have characteristics of interest and then reuse or modify them. Because the administrative tools let measurement developers declare the characteristics of their routines in the registration library, those routines potentially are available to other users of HP TestExec SL. This reusability of measurements and libraries of measurements can be especially beneficial to organizations that have several test systems based on HP TestExec SL.

The Test Development Environment

Overview

The Action Definition Editor and the Test Executive are the main tools used when developing tests and, subsequently, testplans. Various editors and forms appear in the software tools as needed when doing tasks associated with test development. The Test Executive also provides features for debugging tests and running testplans.


Action Definition Editor
1. Write measurement procedures in a standard programming language
2. Create actions from measurement procedures
Test Executive
3. Create tests from actions
4. Create testplans from tests

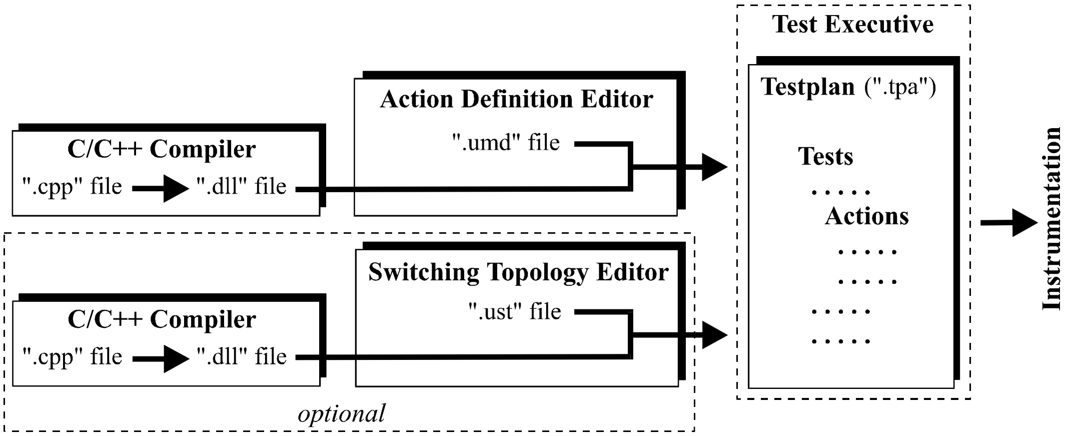
Note

The most important concept to understand here is that developing testplans is a multi-step process that requires multiple tools.

Although it does not appear in the diagram above, some test developers also may use the Switching Topology Editor to define the test system hardware for subsequent use by the Test Executive in controlling switchable connections between resources and the unit under test, or UUT.

How the Software Tools Interact

A high-level, conceptual example of how the software tools interact is shown below.



The upper sequence shows how you can use a C/C++ compiler to create a source file (“.cpp”) containing one or more functions that, when compiled into an executable dynamic-link library (“.dll”), do useful tasks, such as set up a power supply or make a measurement. Collectively, these functions are called “action code” or “action routines.”

However, the Test Executive initially knows nothing about the DLL. You must use the Action Definition Editor to create an “action definition” that associates the DLL with descriptive information needed to use it. Each action definition resides in a “.umd” file that, when used with the DLL, forms an executable component called an “action.”

As the lower sequence shows, you also can create a DLL for an optional component called a “hardware handler,” which is a software layer that enhances HP TestExec SL's interaction with hardware, such as a relay matrix in a switching module. In this case, a hardware handler contains functions the Test Executive calls to interrogate and control switching hardware.

The output from the Switching Topology Editor is a “.ust” file that describes a layer of connections called “switching topology” associated with system resources, connections between the test system and the UUT, or the UUT itself. Used together, a hardware handler for a switching module and a “.ust” file provide information the Test Executive needs if you wish to use its graphical features to control switchable connections during testing, such as connecting the UUT to power supplies, sources, and detectors.

The testplan, which resides in a “.tpa” file, is where these individual elements are used together. A testplan is an ordered sequence of tests that each contains one or more of the predefined actions described earlier. Running the testplan executes tests containing actions that do tasks to test the UUT.

As the testplan runs, actions in its tests control whatever instrumentation is part of the test system. In a specific sense, “instrumentation” may mean a DMM or a frequency counter, but in a more general sense it also includes switching modules, or any other hardware needed to test the UUT.

About System Integration

Traditionally, the system integrator has been the person who assembles, or "integrates," hardware and software to create a complete test system. Such integration might include choosing the hardware, wiring and configuring it, and writing the software needed to use the hardware for production testing. In many ways, a test system assembled this way is a custom creation.

Another necessary role when creating a test system is that of the test developer. A test developer creates and debugs tests used to verify or characterize the operation of whichever kind of unit or module is being tested. For example, one or more test developers might develop a series of tests and then release them for use by a manufacturing line in a production environment containing many test systems.

When you use HP TestExec SL as the basis for a test system, some of the tasks you do can be considered "standard" or mandatory insofar as they always must be done before the test system can be used. Examples of these kinds of tasks include making the connections between the system hardware and the unit under test—often referred to as "fixturing"—and using the Test Executive to write tests that control the hardware.

Other kinds of tasks might be considered truly custom or optional because they are not necessarily done by most system integrators. Examples of these include customizing the operator interface and writing hardware handlers. These custom tasks tend to be more complex than standard tasks, and may require specific skills beyond those needed to do the standard tasks.

You probably will need to do the standard tasks before you can use the test system, while the optional tasks may not be necessary or can be done later when you are more familiar with the test system. Or, you may want to distribute the system integration tasks—such as separating the system integrator and test developer roles—and have each person read the appropriate topics in HP TestExec SL's documentation.

2

Installing & Running HP TestExec SL

This chapter describes how to install and run HP TestExec SL on the Windows 95 and Windows NT platforms.

See Chapter 6 in the *Using HP TestExec SL* book for system administration topics.

Installing & Running on Windows 95

System Requirements

The hardware and software needed to install and run HP TestExec SL on Windows 95 are:

- IBM-compatible PC (486 or faster) with at least 16 MB of RAM
- CD-ROM drive
- 1024 x 768 graphics
- At least 100 MB of free hard disk space (approx. 35 MB for the HP TestExec SL software)
- Microsoft Windows 95

Notes About Installing a New Version Over an Old Version

1. HP TestExec SL stores various configuration settings in its initialization file, “<*HP TestExec SL home*>\bin\tstexcsl.ini”. Because this initialization file may contain custom settings you wish to keep, a new installation of HP TestExec SL does not replace an existing initialization file, nor does it modify its contents.

   However, this conservative approach to updating means that new features that require additional entries in the initialization file are not automatically added. Thus, you must add them manually as follows:

   Because newer versions of HP TestExec SL may include new features that require entries in the initialization file, we recommend that you do the following prior to installing a new version of HP TestExec SL over an old version:

   - Move the existing initialization file to a different location.

- Install the new version of HP TestExec SL.
- Use a text editor, such as WordPad in its text mode, to compare the contents of the old and new initialization files. If the old file has different configuration settings that you wish to keep, add them to the new file.

2. If you wish to keep an older version of HP TestExec SL on your system when installing a new version, install the new version to a new location to keep if from overwriting existing files. Then edit the “[HP TestExec SL]” entry in the “win.ini” file, which is located wherever you installed Windows, to identify which version of HP TestExec SL to run.

Note If you do a Custom installation below and install the original configuration files, you can find a default copy of the initialization file, the datalogging format files, and others in directory “<*HP TestExec SL home*>\DefaultConfiguration”.

To Install the Software on Windows 95

Note The installation CD-ROM supplied with HP TestExec SL also supports installing HP TestExec SL across a network to any disk visible within Windows, such as to a shared disk on another PC.

1. Insert the HP TestExec SL CD-ROM in your CD-ROM drive.
2. Open the Windows 95 Control Panel (located by default in the “My Computer” folder).
3. Choose the Add/Remove Programs icon.
4. Choose the Install button.
5. Follow the instructions that appear and specify the “setup.exe” program on the CD-ROM as the installation program to use.

6. Follow the instructions that appear when the installation program runs.

Note If you will be developing actions in HP BASIC for Windows, choose the Custom installation option and install the modules needed by HP BASIC for Windows. If you wish to have default copies of the various configuration files used by HP TestExec SL available, choose the Custom installation option and install them.

HP VEE Considerations

HP TestExec SL communicates with HP VEE via a high-speed Remote Procedure Call (RPC) mechanism based on the industry-standard Transport Control Protocol (TCP). This is the same mechanism used by HP VEE to invoke other HP VEE servers when using the Import Library object to load remote functions. Because the communication between HP TestExec SL and HP VEE is transparent from a user’s perspective, your only task is to write the HP VEE function library itself.

Before HP TestExec SL and HP VEE can work together, you must have the HP VEE Service Manager program, *veesm*, running on your PC. You probably will want to place a shortcut to this program in your “Startup” group to make it start automatically with Windows 95.

Beside having the Service Manager running, you need an entry for *veesm* in your "\windows\services" file, which might look like this:

```
veesm 4789/tcp # HP VEE service manager
```

See the HP VEE documentation for details.

HP BASIC for Windows Considerations

When you install HP TestExec SL, it looks for HP BASIC for Windows on your system. If it finds HP BASIC for Windows, HP TestExec SL installs additional files needed to develop actions in HP BASIC for Windows.

Given the above, *if you install HP BASIC for Windows after installing HP TestExec SL you will be missing some files that you need*. In this case, simply do a partial reinstallation of HP TestExec SL via the Custom installation option and specify that only the HP BASIC for Windows files should be installed.

To Run HP TestExec SL on Windows 95

1. Launch HP TestExec SL from its folder in the taskbar's Start menu.

2. When HP TestExec SL starts, it prompts you to log in. You must provide a valid login name and password.

   Type "administrator" in the Name field. It requires no password.

   Tip: You probably will want to add a password later to provide additional security for your test system.

3. Once you have logged in, you will be presented with a list of groups to which your login belongs. Choose "Developer" from the list.

Note

The personality of the Test Executive's user interface—e.g., test development or production operator—is determined by which login group you use.

Tip: If you want the test development environment to run each time Windows 95 runs, create a shortcut to "<*HP TestExec SL home*>\bin\tstexcsl.exe" and place it in the Windows 95 "Startup" folder.

To Uninstall HP TestExec SL on Windows 95

1. Open the Windows 95 Control Panel (located by default in the "My Computer" folder).

2. Choose the Add/Remove Programs icon.

3. In the list of applications that appears in the Add/Remove Programs dialog box, choose “HP TestExec SL” as the application to be removed.

4. Click the Add/Remove button and follow the instructions that appear.

Installing & Running on Windows NT

System Requirements

The hardware and software needed to install and run HP TestExec SL on Windows NT are:

- IBM-compatible PC (486 or faster) with at least 32 MB of RAM
- CD-ROM floppy disk drive
- 1024 x 768 graphics
- At least 100 MB of free hard disk space (approx. 35 MB for the HP TestExec SL software)
- Microsoft Windows NT version 3.51 or later

Notes About Installing a New Version Over an Old Version

1. HP TestExec SL stores various configuration settings in its initialization file, "<*HP TestExec SL home*>\bin\tstexcsl.ini". Because this initialization file may contain custom settings you wish to keep, a new installation of HP TestExec SL does not replace an existing initialization file, nor does it modify its contents.

   However, this conservative approach to updating means that new features that require additional entries in the initialization file are not automatically added. Thus, you must add them manually as follows:

   Because newer versions of HP TestExec SL may include new features that require entries in the initialization file, we recommend that you do the following prior to installing a new version of HP TestExec SL over an old version:

   - Move the existing initialization file to a different location.

- Install the new version of HP TestExec SL.
- Use a text editor, such as WordPad in its text mode, to compare the contents of the old and new initialization files. If the old file has different configuration settings that you wish to keep, add them to the new file.

2. If you wish to keep an older version of HP TestExec SL on your system when installing a new version, install the new version to a new location to keep if from overwriting existing files. Then edit the “[HP TestExec SL]” entry in the “win.ini” file, which is located wherever you installed Windows, to identify which version of HP TestExec SL to run.

Note If you do a Custom installation below and install the original configuration files, you can find a default copy of the initialization file, the datalogging format files, and others in directory “*<HP TestExec SL home>*\DefaultConfiguration”.

Installing the Software

Note The installation CD-ROM supplied with HP TestExec SL also supports installing HP TestExec SL across a network to any disk visible within Windows, such as to a shared disk on another PC.

To Install HP TestExec SL on Windows NT 3.51

1. Insert the HP TestExec SL CD-ROM in your CD-ROM drive.
2. Use the Program Manager to run the “setup.exe” program on the CD-ROM.
3. Follow the installation instructions that appear.

To Install HP TestExec SL on Windows NT 4.0 or later

1. Insert the HP TestExec SL CD-ROM in your CD-ROM drive.

2. Open the Windows NT Control Panel (located by default in the “My Computer” folder).
3. Choose the Add/Remove Programs icon.
4. Choose the Install button.
5. Follow the instructions that appear and specify the “setup.exe” program on the CD-ROM as the installation program to use.
6. Follow the instructions that appear when the installation program runs.

Note If you will be developing actions in HP BASIC for Windows, choose the Custom installation option and install the modules needed by HP BASIC for Windows. If you wish to have default copies of the various configuration files used by HP TestExec SL available, choose the Custom installation option and install them.

HP VEE Considerations

HP TestExec SL communicates with HP VEE via a high-speed Remote Procedure Call (RPC) mechanism based on the industry-standard Transport Control Protocol (TCP). This is the same mechanism used by HP VEE to invoke other HP VEE servers when using the Import Library object to load remote functions. Because the communication between HP TestExec SL and HP VEE is transparent from a user’s perspective, your only task is to write the HP VEE function library itself.

Before HP TestExec SL and HP VEE can work together, you must have the HP VEE Service Manager program, *veesm*, running on your PC. You probably will want to execute this program from your “Startup” group (Windows NT 3.51) or create a shortcut to it in the “Startup” folder (Windows NT 4.0 or later) to make it start automatically with Windows NT.

Beside having the Service Manager running, you need an entry for *veesm* in your "\windows\services" file, which might look like this:

```
veesm 4789/tcp # HP VEE service manager
```

See the HP VEE documentation for details.

HP BASIC for Windows Considerations

When you install HP TestExec SL, it looks for HP BASIC for Windows on your system. If it finds HP BASIC for Windows, HP TestExec SL installs additional files needed to develop actions in HP BASIC for Windows.

Given the above, *if you install HP BASIC for Windows <u>after</u> installing HP TestExec SL you will be missing some files that you need*. In this case, simply do a partial reinstallation of HP TestExec SL via the Custom installation option and specify that only the HP BASIC for Windows files should be installed.

To Run HP TestExec SL on Windows NT

1. Launch HP TestExec SL from the HP TestExec SL group in the Program Manager (Windows NT 3.51) or from its folder in the taskbar's Start menu (Window NT 4.0 or later).

2. When HP TestExec SL starts, it prompts you to log in. You must provide a valid login name and password.

   Type "administrator" in the Name field. It requires no password.

Note You probably will want to add a password later to provide additional security for your test system.

3. Once you have logged in, you will be presented with a list of groups to which your login belongs. Choose "Developer" from the list.

Note The personality of the Test Executive's user interface—e.g., test development or production operator—is determined by which login group you use.

Tip: If you want the test development environment to run each time Windows runs, either execute “<*HP TestExec SL home*>\bin\tstexcsl.exe” in the “Startup” group (Windows NT 3.51) or create a shortcut to it in the “Startup” folder (Windows NT 4.0 or later).

Uninstalling the Software

To Uninstall HP TestExec SL on Windows NT 3.51

1. Open the HP TestExec SL program group.

2. Double-click the “UnInstall” icon and follow the instructions that appear.

To Uninstall HP TestExec SL on Windows NT 4.0 or later

1. Open the Windows NT Control Panel (located by default in the “My Computer” folder).

2. Choose the Add/Remove Programs icon.

3. In the list of applications that appears in the Add/Remove Programs dialog box, choose “HP TestExec SL” as the application to be removed.

4. Click the Add/Remove button and follow the instructions that appear.

3

Concepts

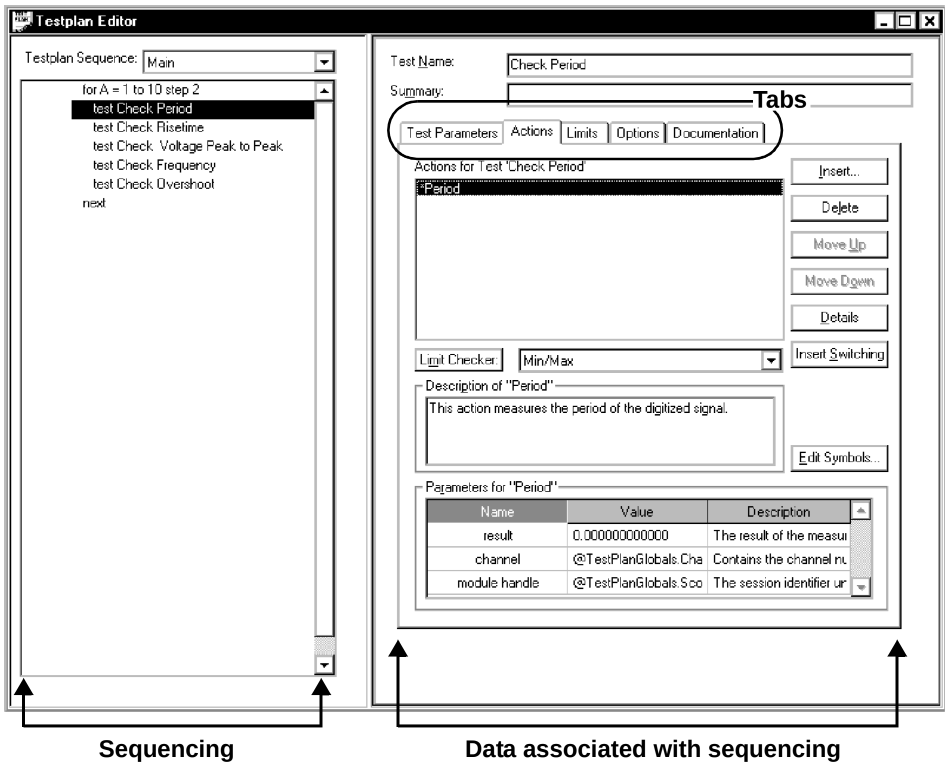
This chapter introduces concepts and terminology you need to understand before using HP TestExec SL. Although features of the software tools are shown here to make you aware of them, the details of their use are described in the *Using HP TestExec SL* book.

Working in the HP TestExec SL Environment

This section provides an overview of several key features of HP TestExec SL's user interface. An awareness of these features is useful because you will encounter them frequently.

Understanding the Relationship Between Tasks & Data

Many of your test development tasks will be done using the Testplan Editor window, which is shown below. Generally speaking, this window is divided into a left pane that contains a sequence of tasks, such as a series of tests to execute, and a right pane that contains data associated with sequenced tasks, such as the details of those tests.



Testplan Sequence: Main

```
for A = 1 to 10 step 2
  test Check Period
  test Check Risetime
  test Check  Voltage Peak to Peak
  test Check Frequency
  test Check Overshoot
next
```

Test Name: Check Period
Summary:

Actions for Test 'Check Period'
*Period

Limit Checker: Min/Max

Description of "Period"
This action measures the period of the digitized signal.

Parameters for "Period"

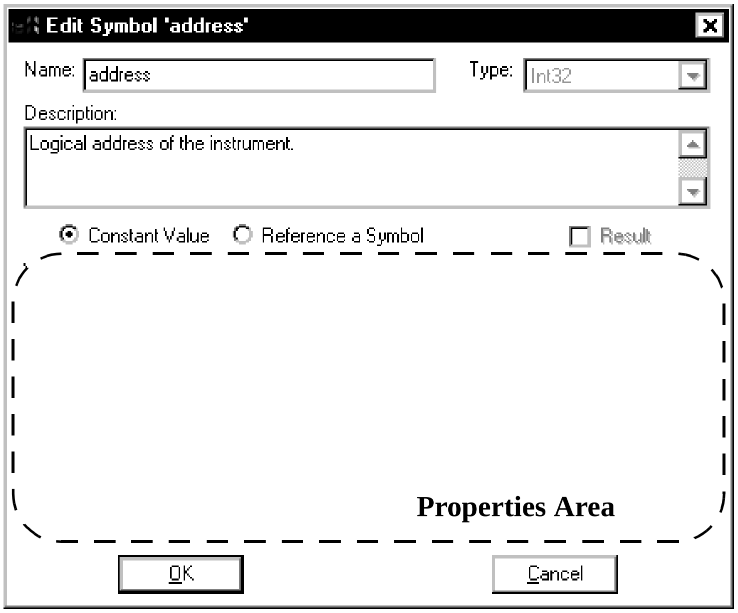
Name	Value	Description
result	0.000000000000	The result of the measu
channel	@TestPlanGlobals.Cha	Contains the channel nu
module handle	@TestPlanGlobals.Sco	The session identifier ur

Because the right pane contains many features, its functionality is organized into logical groups by tabs you can select to display specific subsets of options and data.

Specifying the Properties for Parameters & Symbols

Throughout HP TestExec SL, you will see recurring variations on a generic “properties box” used to specify the characteristics of parameters and symbols. These boxes have titles that begin with Insert or Edit, depending upon the task. Examples include “Insert Symbol,” “Edit Symbol,”, “Insert Parameter,” and “Edit Parameter.” Although you need not understand the details of using these boxes yet, being familiar with them will be useful later.

Although the titles of the boxes varies, their appearance and usage is more similar than different. Each box allows two basic definitions of data: as a value (a constant) or as a reference to a symbol in a symbol table. When used to define or modify values, the box looks similar to this:



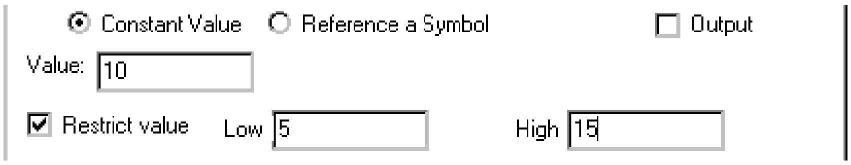
Besides varying with the type of data being described, the appearance of the Properties Area changes when you enable Constant Value or Reference a Symbol.

Features of this box include:

Name	The name of the parameter or symbol.
Type	The data type of the parameter or symbol.
Description	A textual description of the parameter or symbol.
Constant Value	If enabled, the parameter or symbol is a constant whose characteristics are shown in the Properties Area.
Reference a Symbol	If enabled, the parameter or symbol is a reference to a symbol stored in a symbol table, and the characteristics of the reference are shown in the Properties Area.

Result/Output This button's label varies. When it is Result, enabling it means the parameter or symbol is used to determine the pass/fail status of a test. When it is Output, enabling it means the parameter or symbol is used to return a value.

Properties Area A region of the form whose appearance varies with the type of data because different types of data have different attributes. For example, if the data type is Int32 the Properties Area looks like this:

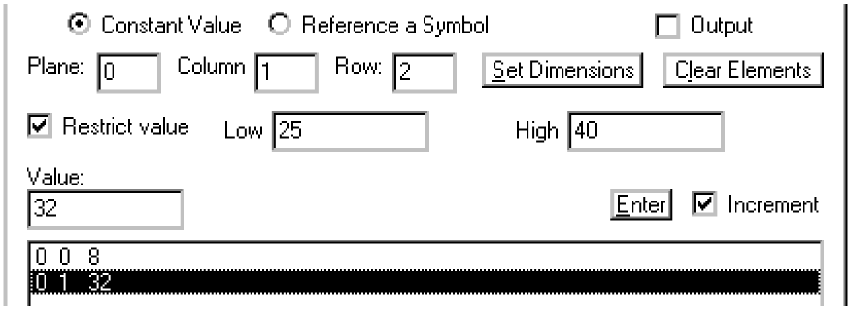


[● Constant Value] ○ Reference a Symbol ☐ Output

Value: 10

[✓ Restrict value] Low 5 High 15

And if the data type is Real64Array, it looks like this:



[● Constant Value] ○ Reference a Symbol ☐ Output

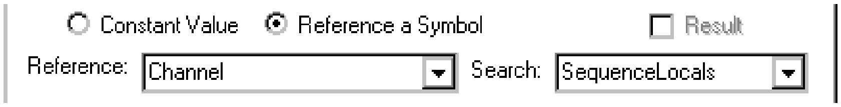
Plane: 0 Column 1 Row: 2 Set Dimensions Clear Elements

[✓ Restrict value] Low 25 High 40

Value: 32 Enter [✓ Increment]

```
0 0  8
0 1  32
```

As shown below, when a parameter or symbol references a symbol table, the Properties Area contains a list that shows the name of the reference and a list of symbol tables to search for the reference.

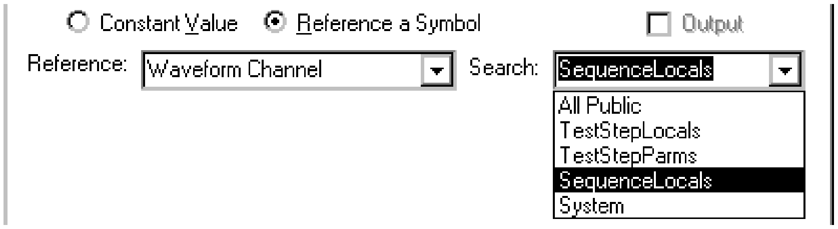


○ Constant Value [● Reference a Symbol] ☐ Result

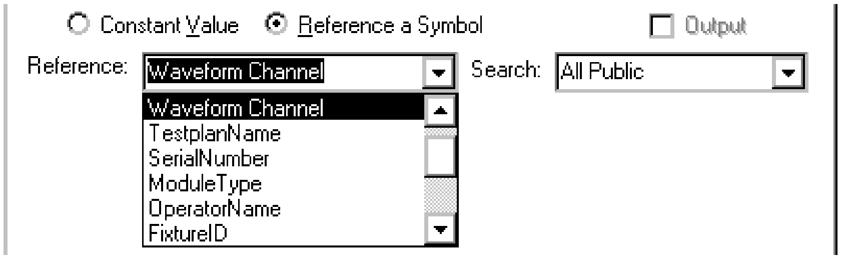
Reference: Channel Search: SequenceLocals

In the example above, the reference is to a symbol named `Channel` in the `SequenceLocals` symbol table.

As shown below, dropping down the Search list shows the names of the symbol tables you can search. If you choose All Public, all the symbol tables in the list are searched, and they are searched in the order in which they appear in the list.



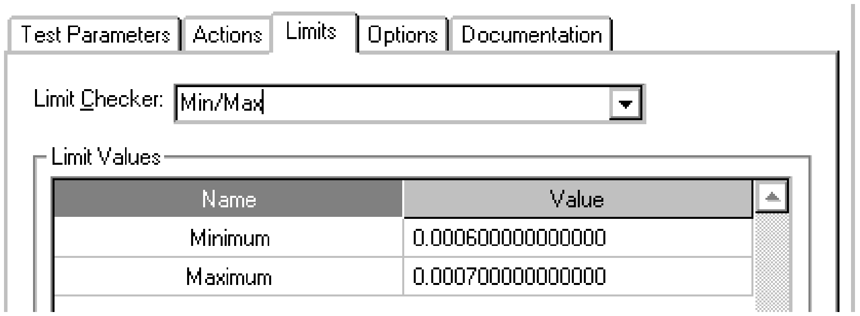
Symbols that are available in the symbol table(s) being searched appear when you drop down the Reference list. The example below shows that when you search All Public, symbols in all the symbol tables (except for external symbol tables) appear in the list.



`WaveformChannel`, which is in the `SequenceLocals` symbol table, once again appears in the Reference list because `SequenceLocals` is among the symbol tables searched when Search is All Public.

Understanding the Two Views of Test Limits

HP TestExec SL provides two ways to view or modify the limits used to decide whether tests pass or fail. The first, which probably is the more straightforward to use, resides on the Limits tab in the right pane of the Testplan Editor window, as shown below.



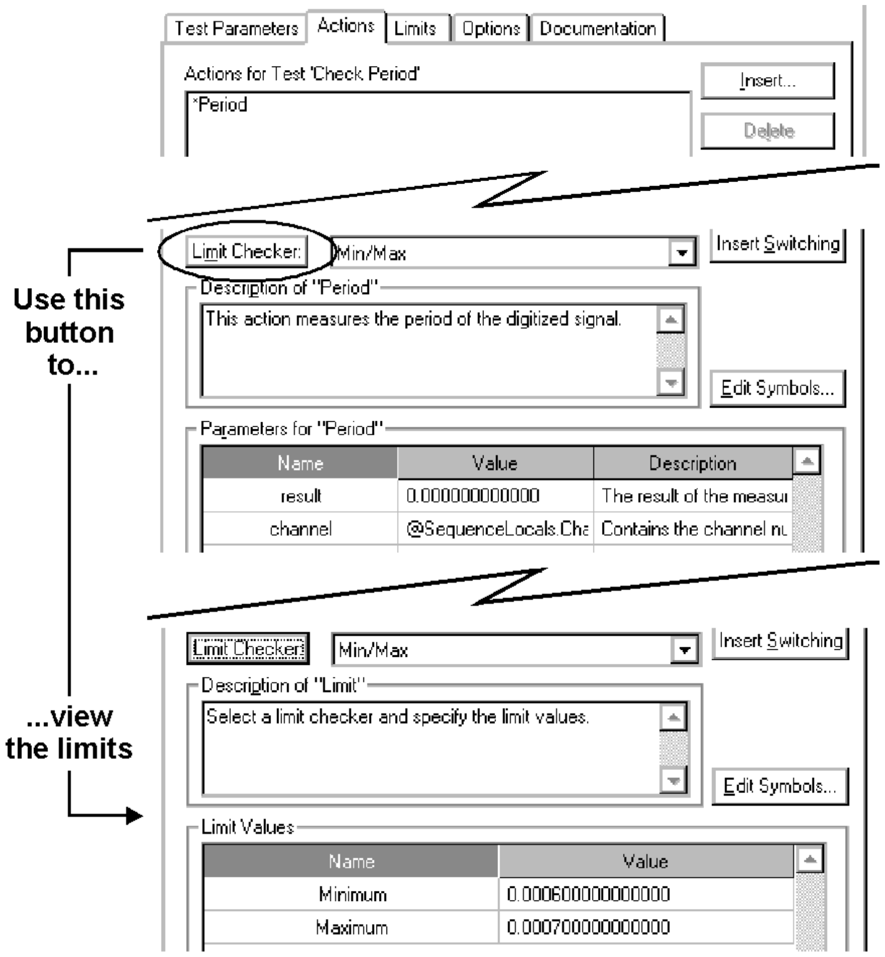
Limit Checker: Min/Max

Limit Values

Name	Value
Minimum	0.000600000000000
Maximum	0.000700000000000

Invoking this view of the test limits is as simple as choosing the Limits tab.

More experienced users may prefer the shortcut provided on the Actions tab, which is shown next. The grid near the bottom of the Actions tab defaults to showing the parameters to actions in the test, but a Limit Checker button lets you quickly switch to viewing the test limits instead. Because this method requires less switching among the tabs in the Testplan Editor window, it can save time when working with a large number of tests.



Actions for Test 'Check Period'

*Period

Description of "Period"

This action measures the period of the digitized signal.

Parameters for "Period"

Name	Value	Description
result	0.000000000000	The result of the measu
channel	@SequenceLocals.Cha	Contains the channel nu

Description of "Limit"

Select a limit checker and specify the limit values.

Limit Values

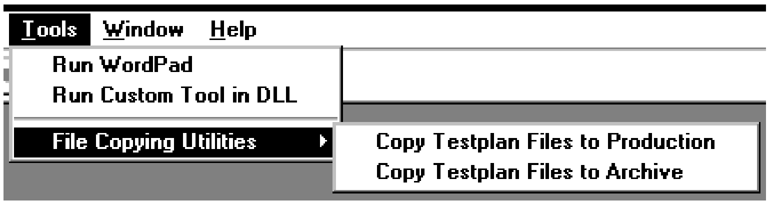
Name	Value
Minimum	0.000600000000000
Maximum	0.000700000000000

Switching to a different tab or clicking an action in the list restores the default view of parameters.

Using Custom Tools to Enhance the Environment

Adding custom tools to the development environment lets you launch programs external to HP TestExec SL or automate repetitive tasks. For example, you could write a batch file that copies a testplan and its related files from a development location to a production location, and then create a custom tool to run that batch file without leaving HP TestExec SL.

As shown below, you can add an optional Tools menu and submenus that let you call executable files or functions in DLLs, plus add separator bars to organize items in menus.

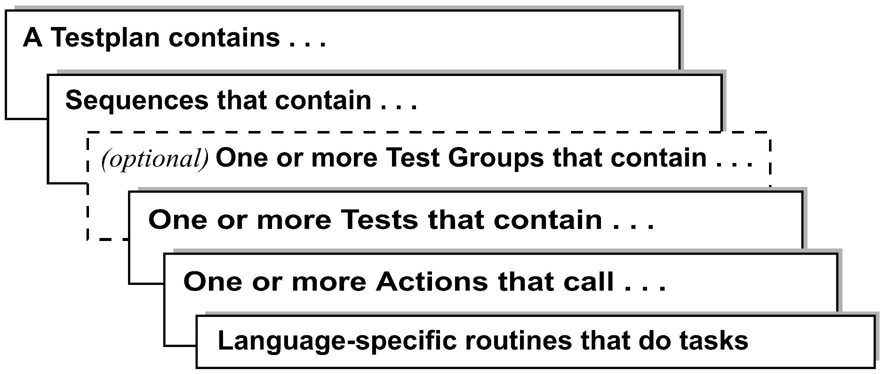


Note A program launched by a custom tool continues running even if you exit HP TestExec SL.

For more information, see “Adding Custom Tools to HP TestExec SL” in Chapter 6 of the *Using HP TestExec SL* book.

About Testplans, Test Groups, Tests & Actions

The hierarchy of the components or building blocks used in the Test Executive environment is shown below.


A Testplan contains . . .
Sequences that contain . . .
(optional) One or more Test Groups that contain . . .
One or more Tests that contain . . .
One or more Actions that call . . .
Language-specific routines that do tasks

These components are:

Testplan	A named entity that contains multiples sequences, or “streams of execution,” used to test a specific device, or UUT (unit under test).
Sequence	A named series of test groups, tests, and flow control statements executed in a predefined order.
Test group	An optional, named set of tests executed in a predefined order. Test groups can be nested inside test groups.
Test	A named sequence of actions executed as a group.
Action	A named call to one or more external routines written in a standard programming language.[a]
Language-specific routines	A routine that does something useful, such as making a measurement.

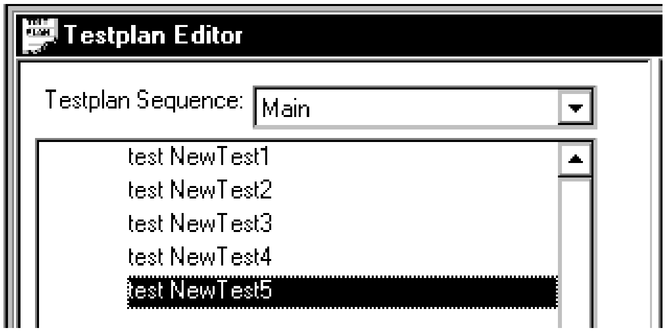
a. The exception to this is a “switching action,” which is built into HP TestExec SL and does not call an external routine.

Although they are not shown above, testplans, sequences, and tests also have named “symbol tables” associated with them. A symbol table is a collection of data items, such as variables, called “symbols” whose usage has a specific scope, such as restricted to a single test or global to an entire sequence of tests. Symbol tables are mentioned where appropriate in this chapter, and further described in Chapter 5 of the *Using HP TestExec SL* book.

A Closer Look at Testplans

What is a Testplan?

As shown below in the left pane of the Testplan Editor, which contains a sequence of tasks, a testplan is a named sequence of tests executed to test a specific unit under test, or UUT. As a testplan contains tests, tests contain actions that call routines that do tasks. This layering of components is used extensively in HP TestExec SL.



```
test NewTest1
test NewTest2
test NewTest3
test NewTest4
test NewTest5
```

A testplan contains multiple streams of execution, or “sequences.” In the example above, the sequence named “Main” contains five tests.

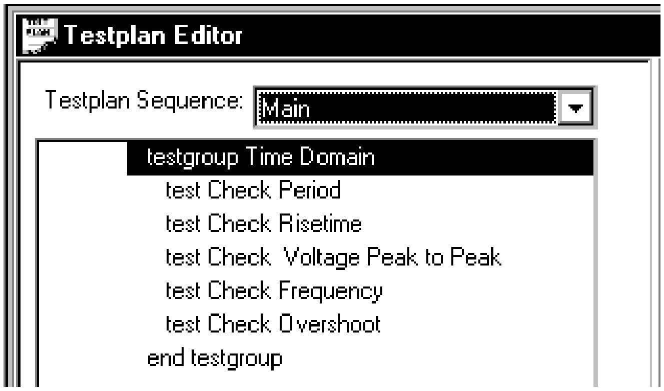
What’s Inside a Testplan?

Test Groups

What is a Test Group?

At its simplest, a test group is an optional, named pair of statements in a testplan. A “testgroup” statement starts the definition of a test group, and an “end testgroup” statement ends the definition. Between these statements you can place test statements, other test group statements, and statements that control the flow of testing, such as “for...next” statements.

An example of the definition for a test group looks like this:



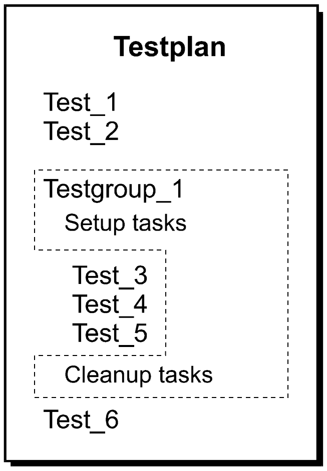
```
testgroup Time Domain
  test Check Period
  test Check Risetime
  test Check  Voltage Peak to Peak
  test Check Frequency
  test Check Overshoot
end testgroup
```

Although the body of this sample test group has only tests in it, it also could contain additional test groups.

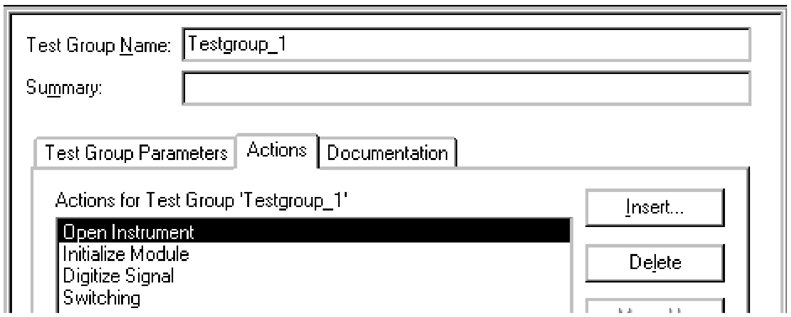
Like tests, test groups can have actions associated with them. What makes a test group unique, though, is that the scope of its actions bounds any tests inside the test group. This lets each test group have an associated list of actions that do tasks before and after the tests inside the group.

Actions in a test group can do common setup tasks needed by all tests in the test group, such as program power supplies to values needed during testing. Also, they can do common cleanup tasks needed after testing ends, such as reset the output of the power supplies to zero.

An example of this is shown below. Here, actions associated with Testgroup_1 do the setup and cleanup tasks for three tests, Test_3 through Test_5. Thus, those tests do not need to duplicate these setup and cleanup tasks. Test groups are a good way to organize a group of tests whose setup and cleanup requirements are alike.



As shown below, you use the right pane of the Testplan Editor window to view or modify the action(s) used to set up or clean up the tests inside test groups.


Test Group Name: Testgroup_1
Summary:

Actions for Test Group 'Testgroup_1'
Open Instrument
Initialize Module
Digitize Signal
Switching

The details of using actions to do setup and cleanup tasks is described in greater detail in “A Closer Look at Actions.”

Why are Test Groups Useful?

Test groups are useful because they let you:

- Organize testplans to do slow actions only once, rather than repeat them in each test.

- Easily and explicitly manage the state of the test system to avoid unnecessary operations.
- Apply a common setup of setup and cleanup tasks to a series of tests.
- Ensure that setup actions are “undone”—i.e., cleaned up—when the test group is exited.

Testplans frequently use nested test groups, where one test group resides within the scope of another. The outer group might set up power supplies, and the inner group set up specific instruments needed by the overall group of tests. For example, a nested test group might set an arbitrary waveform generator to produce a particular waveform, then run the tests that require the waveform.

Sequencing & Flow Control

Within the context of HP TestExec SL, a “sequence” is a named sequence of test groups and/or tests executed in a predefined order. In other words, a sequence is a path of execution through a testplan.

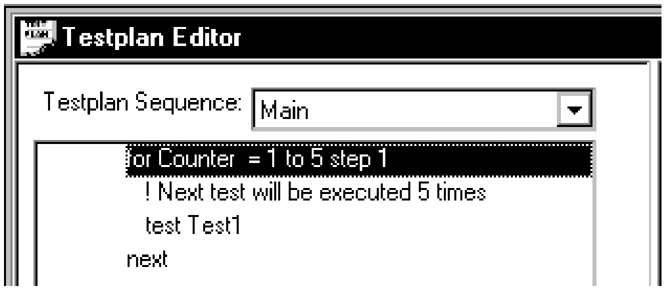
Under normal circumstances, the tests in a testplan execute in the Main sequence that appears as the default sequence in the Testplan Editor’s left pane. However, there may be times when you need more control over the sequence of execution. For example, what should the testplan do if an error occurs during testing? Instead of having the testplan continue, you may want it to branch to an error handling routine, such as a different test that identifies or clears the error condition. Or, you may want to skip subsequent, related tests and branch to an unrelated series of tests.

Even within a given sequence, you may need to control the flow of testing. For example, you may want to loop a specific number of times. Or, you may need to evaluate an expression to decide what happens next in a testplan. And what happens if a test fails? Instead of simply quitting or continuing, you may want to branch to another test that further evaluates the failure.

The next several topics describe features of HP TestExec SL that support multiple sequences of execution and flow control within those sequences.

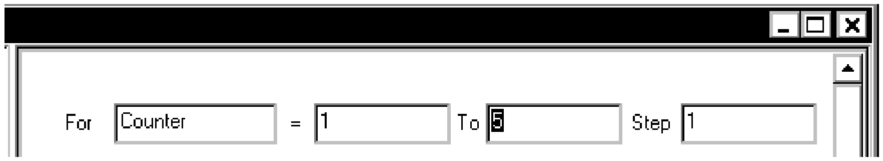
Flow Control Statements

HP TestExec SL supports a variety of BASIC-like flow control statements, such as “If... Then... Else” and “For... To... Step.” As shown below, you can insert flow control mechanisms and descriptive comments directly into testplans.



```
for Counter  = 1 to 5 step 1
  ! Next test will be executed 5 times
  test Test1
next
```

Adding flow control statements is as simple as typing values in predefined “fill in the blanks” forms, an example of which is shown next.



For: Counter = 1 To 5 Step 1

Flow control statements are described in more detail with related tasks in the *Using HP TestExec SL* book.

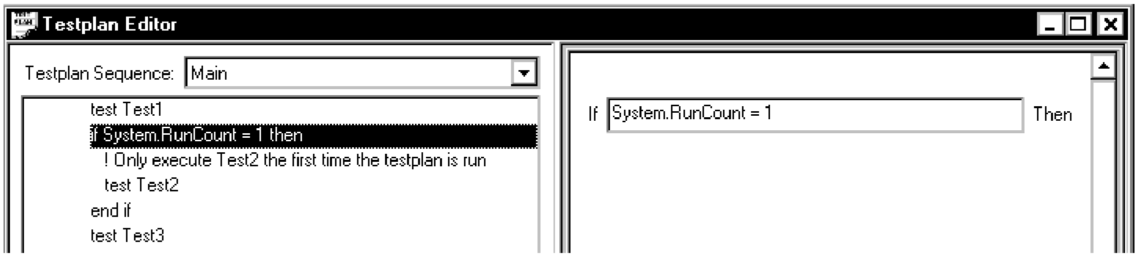
Using Symbols with Flow Control Statements

If desired, you can use a flow control statement to examine or modify the value of a symbol in a symbol table, and then take action based upon the symbol’s value.

Note The syntax for accessing a symbol in a symbol table from a flow control statement is <*symbol table***.** *symbol*>. If you do not specify <*symbol table*>, its value defaults to SequenceLocals.

As an example of using a symbol with a flow control statement, suppose you want one or more tests or test groups to execute the first time a testplan runs but not during subsequent runs. You may be doing this to save time by programming instruments to a known state once per testplan run instead of each time the testplan runs.

As shown below, you can use the value of the predefined RunCount symbol, whose value increments by one each time the testplan runs, in the System symbol table to determine if a test or test group is executed.



```
test Test1
if System.RunCount = 1 then
  ! Only execute Test2 the first time the testplan is run
  test Test2
end if
test Test3
```

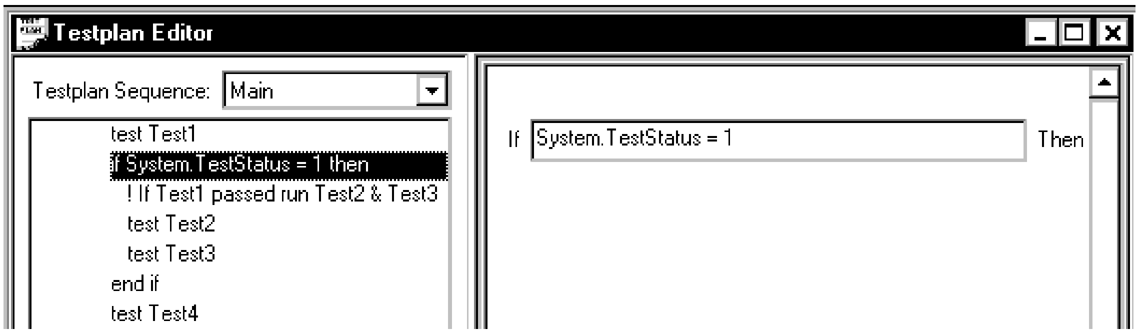
If System.RunCount = 1 Then

In a similar fashion, you can interact with other predefined symbols or symbols that you create from scratch.

For more information about predefined symbols, see “Predefined Symbols in the System Symbol Table” in Chapter 5 of the *Using HP TestExec SL* book.

Branching on a Passing or Failing Test

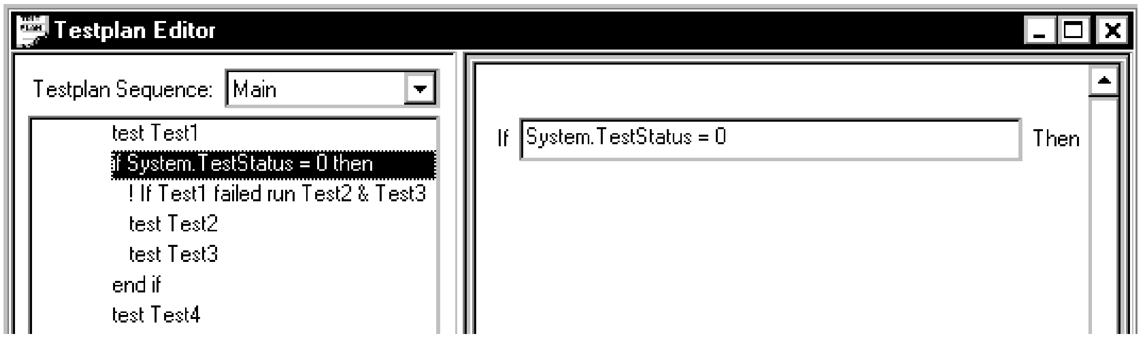
Examining the value of the TestStatus symbol in the System symbol table will tell you whether the most recent test passed (TestStatus = 1) or failed (TestStatus = 0). As shown below, you can use TestStatus in a flow control statement to control the testplan’s flow of execution. This example implements a “branch on pass” feature if the most recent test passed.



```
test Test1
if System.TestStatus = 1 then
  ! If Test1 passed run Test2 & Test3
  test Test2
  test Test3
end if
test Test4
```

If System.TestStatus = 1 Then

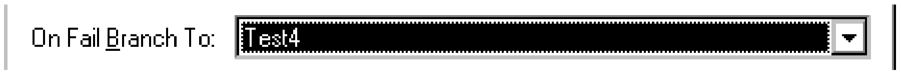
The next example shows how to do the opposite of what was shown above; i.e., implement a “branch on fail” feature by branching if the most recent test failed.



```
test Test1
if System.TestStatus = 0 then
  ! If Test1 failed run Test2 & Test3
  test Test2
  test Test3
end if
test Test4
```

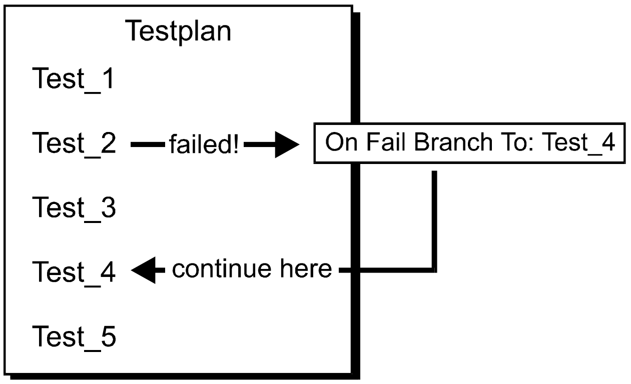
If System.TestStatus = 0 Then

Another way to branch on a failing test is shown below. The Options tab in the right pane of the Testplan Editor has an “On Fail Branch To” feature that lets you branch to a specified label, such as the name of another test or a test group, if the current test fails.



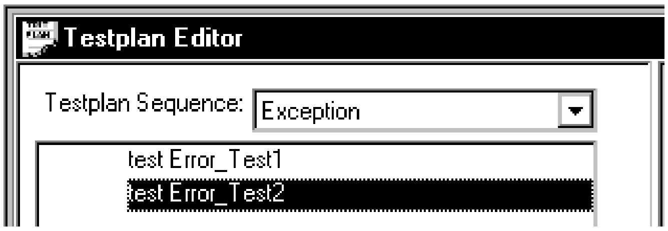
On Fail Branch To: Test4

As shown below, this can be useful for skipping one or more tests if a failure occurs, and then resuming testing elsewhere in the testplan.



Branching on an Exception

Exceptions, which are errors or unusual events that you would not normally expect to happen during testplan execution, can be raised by the underlying code on which the Test Executive is built or by user-defined routines inside actions. When this happens and you do not explicitly handle the exception in the action in which it occurs, the left pane of the Testplan Editor lets you branch to an alternate sequence of tests whose sole purpose is to handle exceptions.

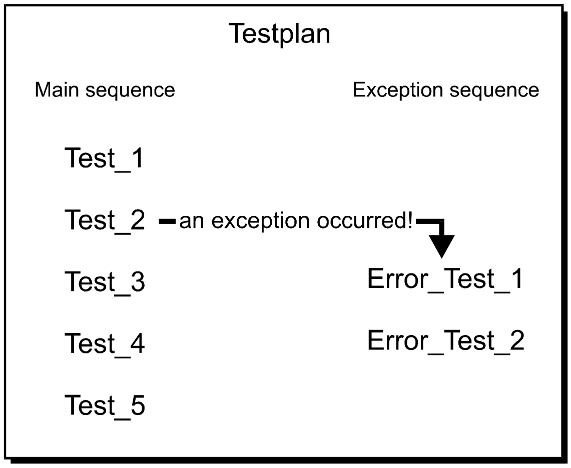


Testplan Editor

Testplan Sequence: Exception

test Error_Test1
test Error_Test2

The example below shows how each testplan actually contains two sequences of execution, one—Main—that is used when tests execute normally and another—Exception—that is used only if an exception occurs. Here, an exception while `Test_2` was executing caused a branch to an alternate sequence of tests called the “Exception sequence” that is used to handle errors.



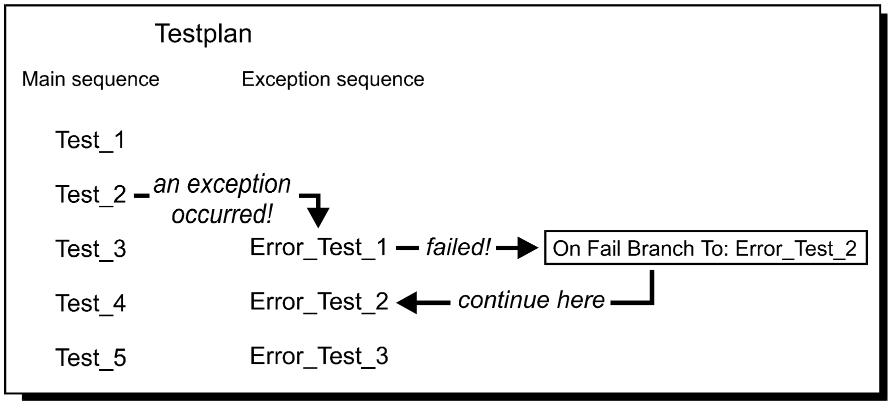
Keep the following in mind when using an Exception sequence:

- Only an exception causes branching to the Exception sequence; i.e., you cannot force branching to the Exception sequence via a failure or other means.

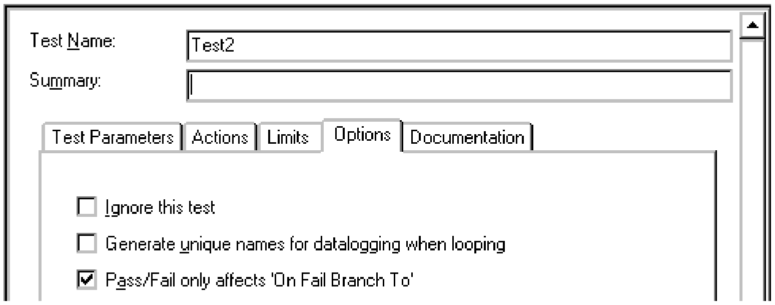
- If you do not want an exception to force branching to the Exception sequence, you must handle the exception in the action in which it occurs.
- All tests share a single Exception sequence; i.e., there is not a separate Exception sequence for each test in the Main sequence.
- When an exception causes branching to handle an error condition, you should assume nothing about the state of the test system or the UUT. Thus, in most cases you should immediately reset everything to a safe state in the test to which you branch. For example, you should immediately discharge capacitors on the UUT, reset signal sources and power supplies, and then reset any remaining instruments.
- The Exception sequence may need to bleed trapped charge from the UUT by programming power supplies and other sources to zero volts and then waiting before resetting the test system.

More Complex Branching

If desired, you can create more complex testplans by using combinations of branching on failures and branching on exceptions. As shown next, a test in the Exception sequence can branch on a failure the same as a test in the Main sequence. However, its branching is restricted to other tests in the Exception sequence; i.e., it cannot branch to a test in the Main sequence of tests.



If desired, you can use the “On Fail Branch To” feature—i.e., branching on a failure—to force branching by creating a test that returns a result, setting the test's pass/fail limits so that it always fails, and not setting the test's Save Pass/Fail option in the right pane of the Testplan Editor (shown below).



Test Name: Test2
Summary:

☐ Ignore this test
☐ Generate unique names for datalogging when looping
☑ Pass/Fail only affects 'On Fail Branch To'

Testplan Variants

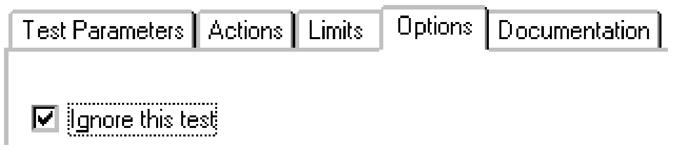
Testplans support the use of one or more named variations called "variants" that can define the behavior of the tests and test groups inside them. Because they let you use *one* testplan with *n* different sets of test limits and parameters, variants are useful where one UUT is very similar to another except for slightly different values for its test limits or parameters.

If a testplan has variants[1], you can do the following for each of its tests and test groups:

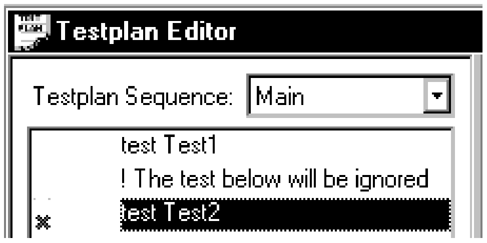
- Choose a variant under whose name you wish to define a distinct version of the test or test group.
- Define a set of parameters and limits for a version of a test associated with a variant.
- Execute or ignore each test by variant.

1. All testplans have a default variant named "Normal".

Only tests that you specifically identify to be ignored for a variant, as shown below, are not executed.



☑ Ignore this test



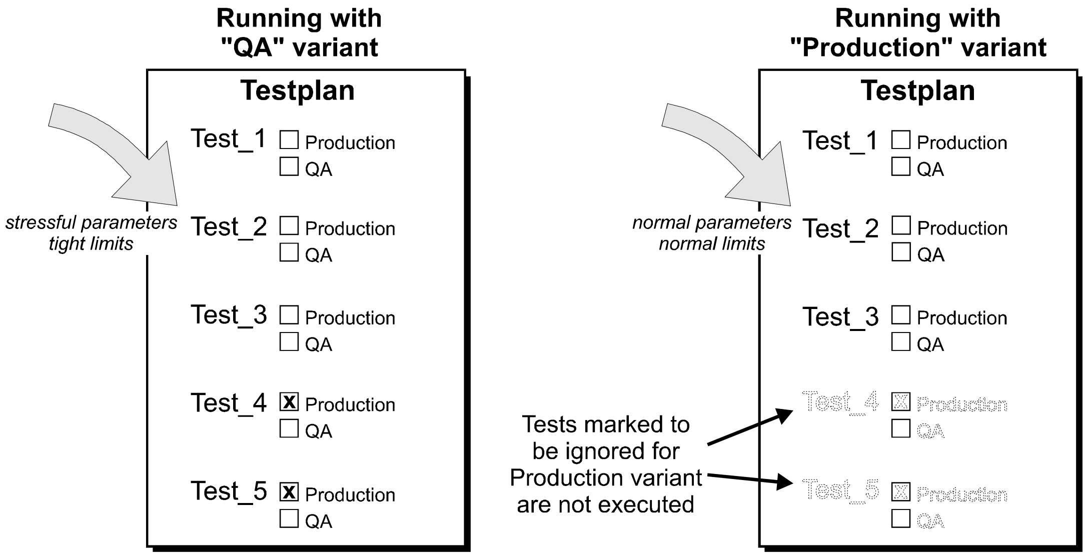
Testplan Sequence: Main

```
      test Test1
      ! The test below will be ignored
✖     test Test2
```

Note The most important thing to know about variants is that they can provide a single testplan with multiple personalities.

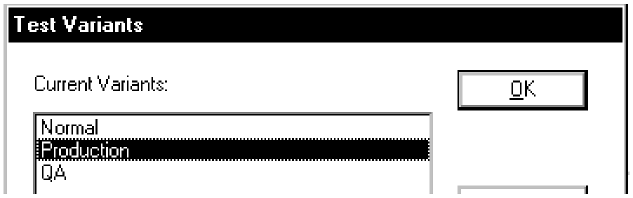
When the testplan is run, you specify which variant to use. This is similar to having different versions of a testplan available, except that what appears to be different versions is actually multiple views of the same testplan dependent upon which version of the tests in it are executed or ignored based on variants.

The following example shows the use of two variants, QA (Quality Assurance) and Production, in a testplan. Check boxes to the right of each test indicate which tests are ignored for the variants. Notice that every test is executed for the QA variant but that tests 4 and 5 are ignored for the Production variant.


Running with "QA" variant
Running with "Production" variant
Testplan
Test_1 ☐ Production ☐ QA
Test_2 ☐ Production ☐ QA
Test_3 ☐ Production ☐ QA
Test_4 ☒ Production ☐ QA
Test_5 ☒ Production ☐ QA
stressful parameters tight limits
normal parameters normal limits
Tests marked to be ignored for Production variant are not executed

When the testplan is run with the QA variant, all the tests are executed and the stringent set of parameters and limits associated with the QA variant is passed to them. But when the testplan is run with the Production variant, two fewer tests are executed and the normal parameters and limits appropriate for production testing (associated with the Production variant) are passed to the tests.

Various features in the Test Executive let you define variants for tests, and then specify which variant to use when executing a series of tests in a testplan, respectively.


Test Variants

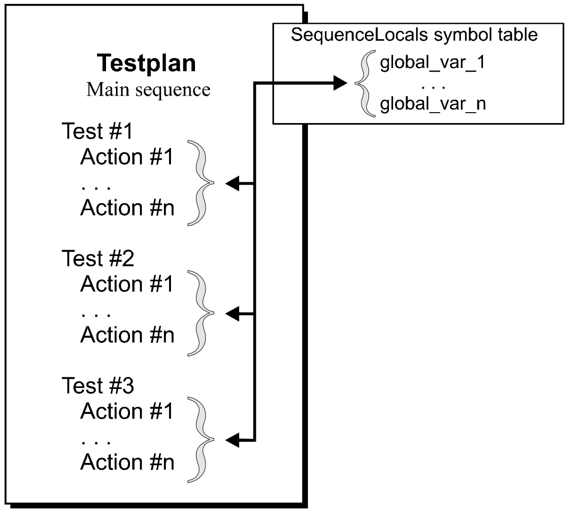
Current Variants:

Normal
Production
QA

Also, you can associate the switching or setup/cleanup actions in test groups with variants used in the testplan. Unlike tests, however, you cannot execute or ignore test groups based on variants.

Global Variables in Testplans

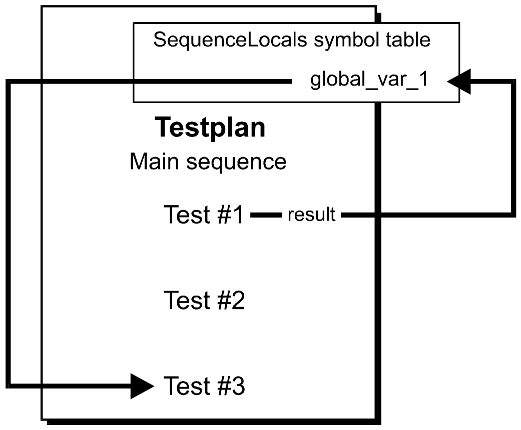
If desired, you can define global variables—i.e., variables whose scope is the testplan—in a testplan. Global variables let you share data among the actions in all the tests in a testplan. As shown below, one place global variables can be stored is in the SequenceLocals symbol table,[1] and they are accessible as parameters for any action in any test included in the Main sequence.



The example below shows the result from Test #1 being passed to Test #3 via a global variable named *global_var_1*. In this simple example, the result from Test #1 might be a value needed in Test #3. When Test #1 finishes, the

1. Note the distinction between this and the *TestStep*Locals symbol table used to pass results between actions inside a single test.

result is passed by reference to *global_var_1*. When Test #3 executes, it is passed the value of the global variable as one of its parameters.



In a similar fashion, the value returned by Test #1 is visible to the entire testplan. Thus, Test #1 could return a value—a baud rate, perhaps—used in any or all subsequent tests.

Besides the SequenceLocals symbol table used to store global variables for the Main sequence, there is another symbol table named SequenceLocals that stores global variables whose scope is tests included in the Exception sequence. Although both tables have the same name, each instance of it is uniquely accessible only in the sequence in which it appears.

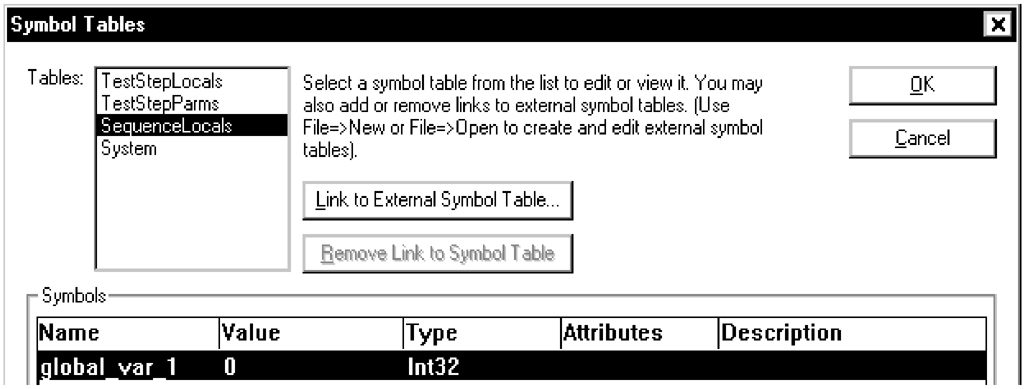
If you need a symbol table whose scope is the entire testplan—i.e., all the tests and actions in both the Main sequence and the Exception sequence—you can either use the TestPlanGlobals symbol table or create an external symbol table whose symbols are stored in an external file associated with the testplan.[1]

External symbol tables can be useful when you wish to support multiple versions of a testplan. For example, each symbol table can hold the data (symbols and their default values) used to define a specific version of the testplan. Also, you can name an external symbol table whatever you like.

1. There also is a symbol table named System whose scope is the entire testplan. It contains predefined symbols associated with the testing environment.

Note

Be aware that the SequenceLocals symbol table contains no predefined variables. You must use the Symbol Tables box, which is shown below, to add a new variable to the symbol table.



Select a symbol table from the list to edit or view it. You may also add or remove links to external symbol tables. (Use File=>New or File=>Open to create and edit external symbol tables).

Name	Value	Type	Attributes	Description
global_var_1	0	Int32		

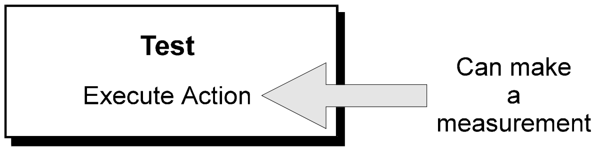
A Closer Look at Tests

What is a Test?

A test is a named procedure that does some form of testing activity on a unit under test, or UUT. To be meaningful, most tests have a limits checking feature that determines if the UUT passed or failed the test. Also, most tests use a datalogging feature to store information collected during the test, such as information about failing tests for subsequent analysis.

What's Inside a Test?

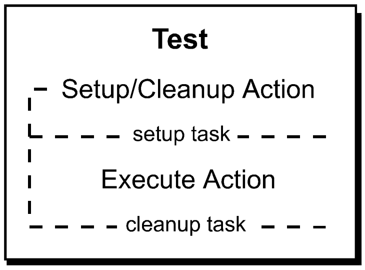
A test consists of one or more actions and their associated data executed in a predefined sequence. An action calls an action routine, which is custom code you write that does something useful, such as making a measurement. An action called an "execute action" forms the basis for a test, as shown below.



Note Although actions and the routines they call are separate components, it is usually simplest to refer to them collectively as "an action." Thus, we might say "an execute action makes a measurement" and actually mean "an execute action calls an action routine that makes a measurement."

An execute action may be all that is needed for a test. But there are times when it is useful to have other tasks precede or follow an execute action. For example, you may need to set up the conditions for a test—i.e., close relays that make necessary connections, set power supplies to known values, set a DMM to a specific range, etc.—before using an execute action to make the measurement.

To address this need, you can include a "setup/cleanup action" that precedes the execute action in your test. A setup/cleanup action can have a setup component that executes an action routine before the execute action begins, and a cleanup component that executes another action routine after the execute action ends.



If desired, you can use more than one of each kind of action in a test. For example, you might have several setup/cleanup actions whose setup components establish the initial conditions for the test. After that, an execute action might make a preliminary measurement—to return an offset voltage, perhaps—followed by a second execute action that makes the measurement used to decide if the test passes or fails. Finally, the cleanup components of the setup/cleanup actions might restore the hardware to a known state.

A third type of action, called a “switching action,” lets you close connections, such as switching paths made with relays, at the beginning of a test and controls the status of those connections when the test ends.

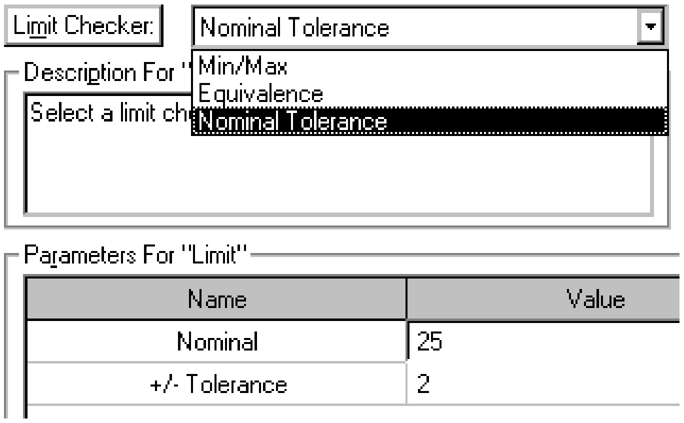
Actions are described in greater detail under “A Closer Look at Actions.”

Limits Checking

To be meaningful, most tests have a limits checking feature that determines if the UUT passed or failed the test. Limits define the acceptable boundaries for a test. If the results from a test are outside its specified limits, the test fails. If desired, a test can have more than one set of limits, where each set is associated with a named variant of a testplan, such as “Hot” or “Cold.”

Whenever you use the Testplan Editor’s right pane to define a test that includes an action that returns a result, you have the option of specifying limits for the test.[1] Conceptually, you can consider limits checking as a feature that is built into the framework of each test.

As shown below, HP TestExec SL includes several kinds of limits checkers.



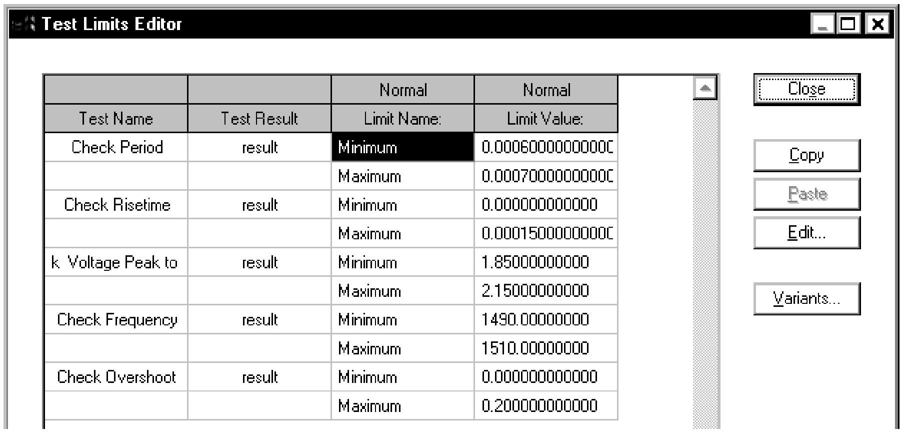
Name	Value
Nominal	25
+/- Tolerance	2

The limits checkers you can use are:

Min/Max	The test passes if its result is within specified minimum/maximum values. To pass, the result must be greater than or equal to the minimum and less than or equal to the maximum.
Equivalence	The test passes if its result exactly matches a specified value.
Nominal Tolerance	The test passes if its result is within a specified +/- tolerance of a specified nominal value. To pass, the result must be greater than or equal to the lower tolerance value and less than or equal to the upper tolerance value. *Note:* The nominal tolerance is a simple number and not a percentage. In other words, a nominal tolerance of 5 means the result must be within plus or minus 5 of the nominal value, not within plus or minus 5 percent of it.
<No Limits>	The test is not checked against pass/fail limits.

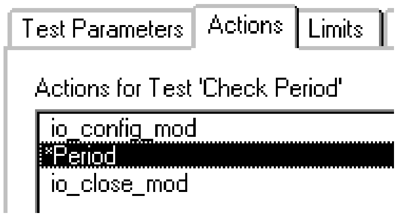
1. If you do not specify limits, they assume a default value assigned when the action was created. The default may or may not be what you need.

If desired, you can also use the Test Limits Editor box shown below for a global perspective that lets you examine or modify test limits across an entire testplan.



		Normal	Normal
Test Name	Test Result	Limit Name:	Limit Value:
Check Period	result	Minimum	0.00060000000000
		Maximum	0.00070000000000
Check Risetime	result	Minimum	0.000000000000
		Maximum	0.00015000000000
k Voltage Peak to	result	Minimum	1.85000000000
		Maximum	2.15000000000
Check Frequency	result	Minimum	1490.00000000
		Maximum	1510.00000000
Check Overshoot	result	Minimum	0.000000000000
		Maximum	0.200000000000

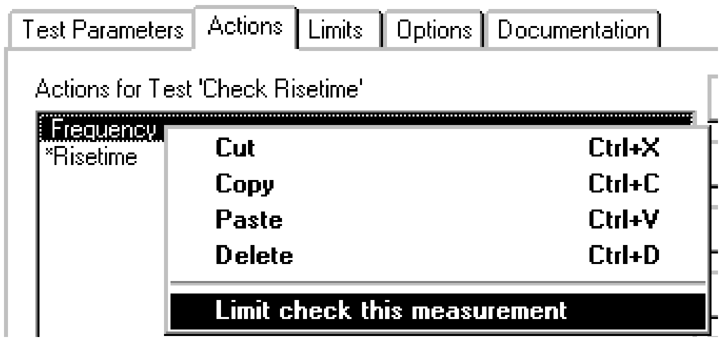
The list of actions that appears in the right pane of the Testplan Editor window shows an asterisk preceding the action whose result is used for limits checking. As shown below, if a test contains various kinds of actions, the action whose name has an asterisk beside it need not necessarily be the last action in the list.



Actions for Test 'Check Period'

```
io_config_mod
*Period
io_close_mod
```

If a test contains more than one execute action, you can specify which action is used for limits checking. The next example shows changing the action used for limits checking from Risetime to Frequency.

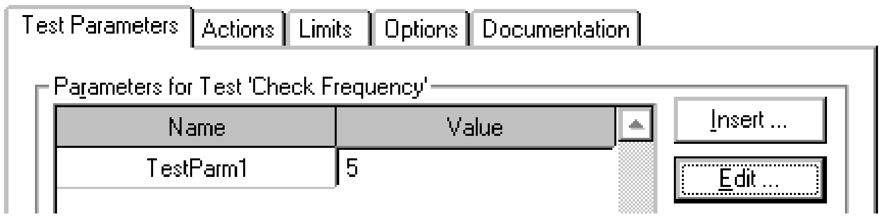


Parameter Passing

Tests contain actions that do something useful, such as make a measurement. In a sense, each test is generic until parameters are passed to it and to its actions to create a specific instance of the test. For example, a generic test that programs a power supply to produce an output, programs a voltmeter to a range, and makes a voltage measurement must be passed specific values for the power supply and voltmeter, and pass/fail limits for the measurement. Thus, a test is a “base” specification whose “overrides” determine its final characteristics.

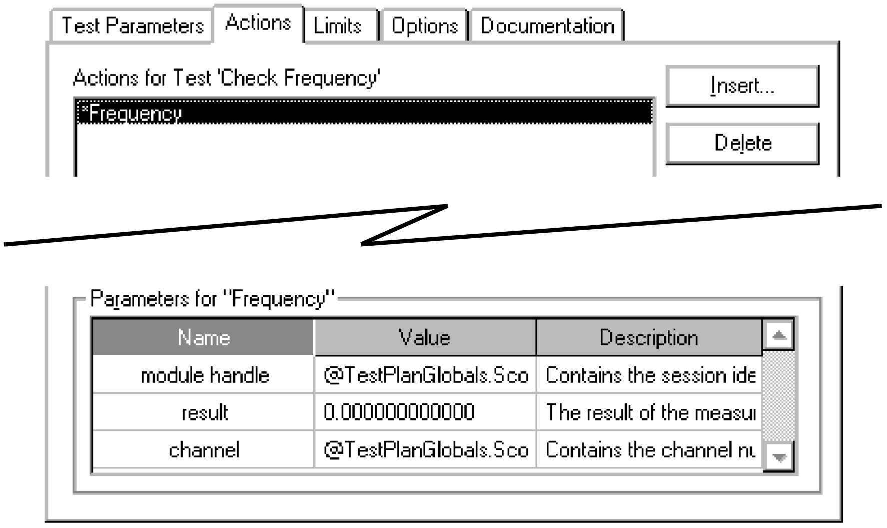
Besides passing data in parameters to a test, you can also pass a value that determines if or how the test is executed. For example, suppose two tests were named Test1 and Test2. Depending upon the results from Test1, you could pass a value from it to a global variable in a symbol table accessible to both tests. The value of the global variable could then be passed into Test2 and evaluated to determine whether Test2 executes.

Shown below is how the Test Parameters tab in the right pane of the Testplan Editor lets you specify parameters for tests.



Name	Value
TestParm1	5

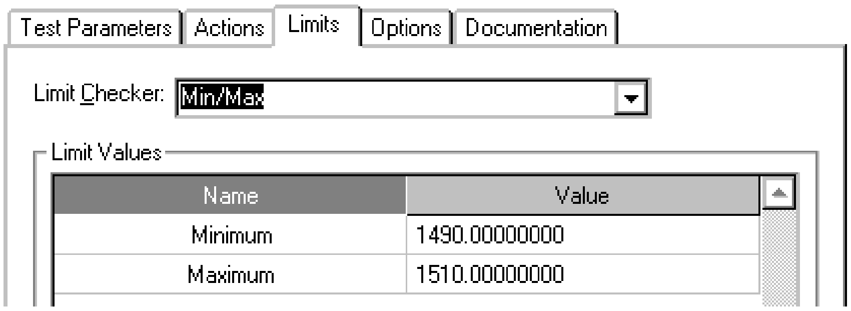
Also, each action in a test has a list of parameters, and each parameter in that list has a default value defined when the action was created. As shown below, you use features at the top and bottom of the right pane in the Testplan Editor to specify the values for parameters in actions that the Test Executive uses to create a specific instance of the test when it executes the test.



Name	Value	Description
module handle	@TestPlanGlobals.Sco	Contains the session ide
result	0.000000000000	The result of the measu
channel	@TestPlanGlobals.Sco	Contains the channel nu

Notice that the values of two of the parameters shown above—“module handle” and “channel”—begin with @. This indicates that instead of passing values directly, the parameters reference symbols in symbol tables.

In a similar fashion, the Limits tab in the Testplan Editor's right pane lets you specify pass/fail limits that help define a unique instance of a test. An example is shown below.



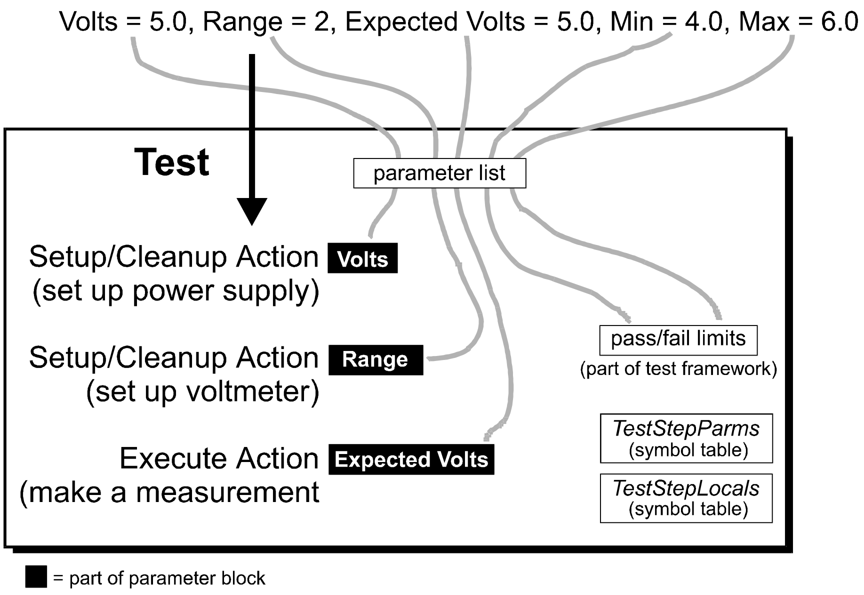
Limit Checker: Min/Max

Limit Values

Name	Value
Minimum	1490.00000000
Maximum	1510.00000000

Note The values of parameters that return numeric results used for limits checking are converted to and displayed as reals.

As shown below, parameters and pass/fail limits (which are a part of the test framework) passed into a test define a unique, specific test that does a specific task. Parameters are passed in a named group called a "parameter block,” which is explained in greater detail in the *Using HP TestExec SL* book.



Each test also contains a symbol table named TestStepParms used to hold values passed as parameters to the test (as opposed to parameters passed to actions in the test). Also, a symbol table named TestStepLocals stores symbols whose scope is the test.

Although this example is simplistic in the sense that actual tests probably will be passed many parameters instead of only a few, the concept is the same in either case.

A Closer Look at Actions

What is an Action?

An “action” is the smallest component of a test. It is a routine that does something useful, such as making a measurement or controlling the switching operations needed for a particular test.

There are three types of actions:

execute	Does a task, such as make a measurement
setup/cleanup	Has optional *setup* and *cleanup* components that can do tasks before and after an execute action or by themselves
switching	Controls switching hardware

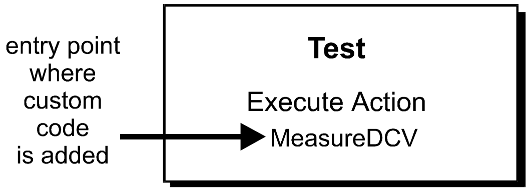
Note Switching actions are described under “How Actions Control Switching.”

In many cases, actions are reusable. For convenience, actions are stored in libraries whose contents you can quickly search. If the action you need already exists, you can copy it and use it as-is in your test. Or, you can modify existing actions or create new ones from scratch if none of the existing actions suits your needs.

For more information about libraries, see “About Test & Action Libraries.“

What’s Inside an Action?

Think of an action as a predefined framework to which you must add a custom action routine—i.e., code that you write—specific to your needs. The action routine is written in a language such as C and then associated with an entry point in the action. When the action is called by a test, it executes the routine associated with it. For example, the execute action shown below calls an action routine named `MeasureDCV` that triggers a digital multimeter to make a DC voltage measurement.

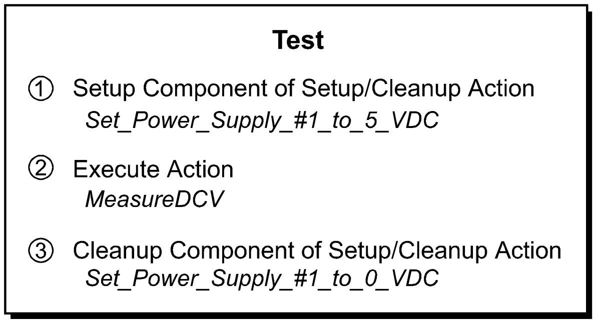


For simplicity, this example and others that follow show pseudo-code notation for the action routine or its purpose. The actual code you add depends upon the means by which you control instruments, such as directly from C/C++ or through an instrument control language such as HP VEE.

Continuing with the example above, suppose you also needed to program a power supply to an initial value before making the measurement, and return the power supply to zero afterward. The sequence of events might be:

1. Program power supply #1 to 5 VDC.
2. Make the DC voltage measurement.
3. Program power supply #1 to 0 VDC.

The actions and custom routines to do this sequence of tasks is shown below, with circles to indicate the order of execution.

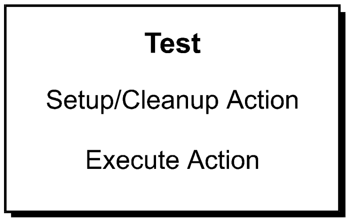

Test
① Setup Component of Setup/Cleanup Action
Set_Power_Supply_#1_to_5_VDC
② Execute Action
MeasureDCV
③ Cleanup Component of Setup/Cleanup Action
Set_Power_Supply_#1_to_0_VDC

The example now includes *one* setup/cleanup action. The setup/cleanup action’s setup component calls an action routine that does a task before the execute action (programming the power supply to 5 VDC), and its cleanup component calls an action routine that does a task afterward (programming the power supply to zero).

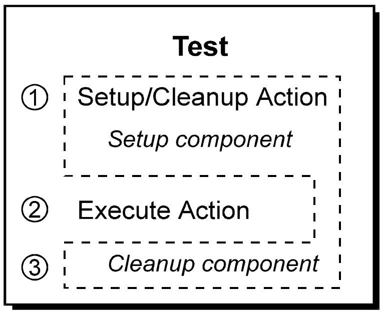
Note Although either of the components in a setup/cleanup action is optional or will work by itself, in most cases you will use them as complementary pairs; i.e., whatever you have the setup component do, the cleanup component will undo.

Paired Structure in Actions

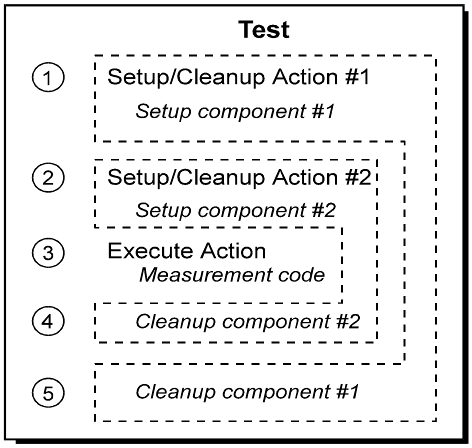
Insofar as the order of execution is concerned, the example above implies that the setup component of a setup/cleanup action is located somewhere before the execute action, and the cleanup component is located somewhere after the execute action. In reality, though, the actions in the test look like this:



How is this possible? As shown below, the order of execution of the setup and cleanup components in a setup/cleanup action is determined by their relationship with the execute action. When a setup/cleanup action precedes an execute action, HP TestExec SL ensures that the setup component executes before the execute action and the cleanup component executes after the execute action.



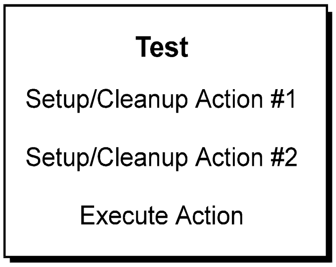
When more than one setup/cleanup action is used in a test, the nesting of the setup and cleanup components is determined by the order in which the actions occur. Refer to the example below.



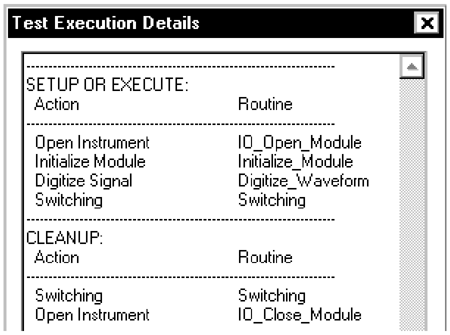
The numbered circles in the example above show how inner loops finish before outer loops, which means the order in which setup and cleanup components in setup/cleanup actions are executed is similar to the operation of loop control structures used in many programming languages.

Note Actions execute in the order specified, so you must be sure to specify them correctly. For example, if the execute action appeared first in the test above, it would execute before either of the setup/cleanup actions.

As mentioned earlier, the order of execution of setup and cleanup components is determined by the relationship between the setup/cleanup action in which they appear and execute actions. Thus, the actions in the previous example really look like this:

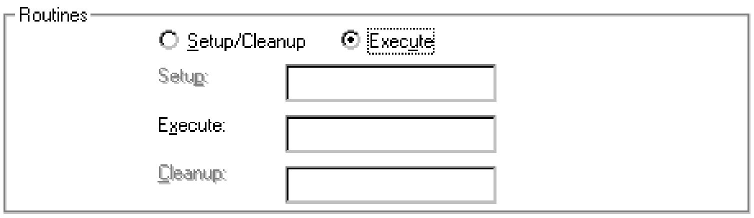


Fortunately, you do not need to remember all the details of the actions and their components inside each test. The Test Executive environment provides a Test Execution Details window, which is shown below, that you can use to view the details of tests, including the sequence in which the components in its actions execute.



```
-----------------------------------------------------------------
SETUP OR EXECUTE:
 Action                         Routine
-----------------------------------------------------------------
 Open Instrument                IO_Open_Module
 Initialize Module              Initialize_Module
 Digitize Signal                Digitize_Waveform
 Switching                      Switching
-----------------------------------------------------------------
CLEANUP:
 Action                         Routine
-----------------------------------------------------------------
 Switching                      Switching
 Open Instrument                IO_Close_Module
```

Also, when you use the Action Definition Editor to create actions, it helps you specify their contents correctly. For example, an execute action cannot have setup or cleanup components, and the editor prevents you from making inappropriate choices. As shown below, the fields for specifying the names of Setup and Cleanup components are disabled when the type of action is Execute.



Which Kind of Action Do You Need?

Given that you have execute actions and setup/cleanup actions to choose between, how do you decide which to use in a given situation? For example, should you use several simple actions, such as a series of execute actions that each do only one task, or something more complex, such as an execute action preceded by more complex setup/cleanup actions?

Keep the following in mind when deciding which kind of action to use in a test:

- If an action does a single, specific task, such as make a measurement, use an execute action.
- If an action has obvious or natural setup and cleanup components, use a setup/cleanup action. For example, you should always use a setup/cleanup action with sources such as power supplies, DACs, etc.

  Complementary setup/cleanup pairs are the best way to ensure that essential actions occur in a specific order. For example, if you program a power supply to some voltage in a test, you probably need to program it to zero when the test has finished. A setup/cleanup action does this well and, because both components are combined in a single action, it ensures they remain together if you move the action in your testplan.
- If an action must handle an exception condition, use an execute action.

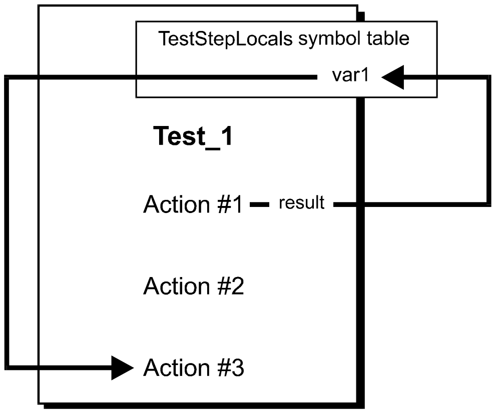
  Suppose you are aware that the UUT has a failure mode that could keep it from responding correctly. Because this represents an exception instead of a catastrophic failure, you want the test to recover gracefully by handling the exception, Here, you should use an execute action that compensates for or clears the failure condition.

Having a strategy for using actions in tests is especially important if you create your own actions because you cannot create appropriate actions unless you can anticipate how they will be used. The simpler an action is, the more likely it is to be reusable. Thus, actions that do a single task are more likely candidates for reuse than more complex actions that do several tasks. For example, an action that does nothing more than program an instrument to a specific range may be reusable in several tests, with only its parameters changed from instance to instance. The trade-off when using simple actions is that you may need several of them to do what one more comprehensive action could do, which increases the complexity of your test.

Passing Results Between Actions Inside Tests

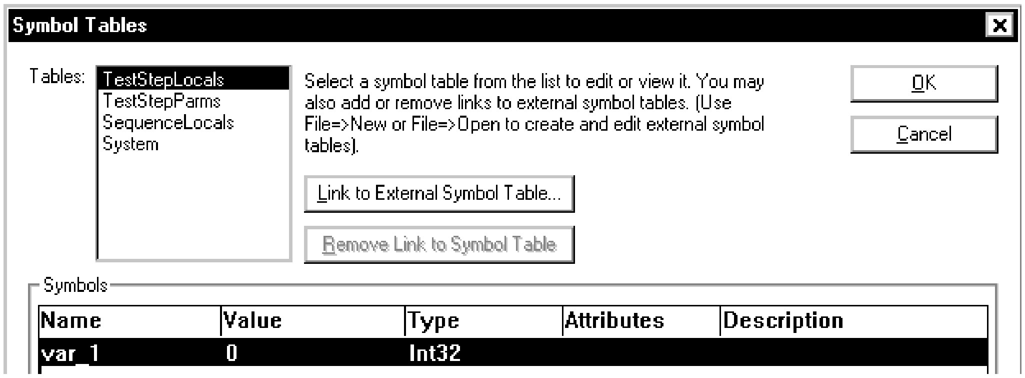
If desired, you can pass the results from one action to another within a test. For example, the result from one action might be passed as a parameter that determines what another action does. Variables defined in the symbol table called TestStepLocals[1], whose scope is the test, are used to pass values between actions inside a test.

The example below shows the result from Action #1 being passed to Action #3 via a variable named *var1*. In this simple example, the result from Action #1 might be an op-amp offset needed as a correction factor in Action #3. When Action #1 finishes, the result is passed by reference to *var1*. When Action #3 executes, it is passed the value of the variable as one of its parameters.



1. Note the distinction between this and the *Sequence*Locals symbol table, TestPlanGlobals symbol table, and external symbols tables used to pass results at the testplan level.

Note Be aware that the TestStepLocals symbol table contains no predefined variables. You must use the Symbol Tables box, which is shown below, to add a new variable to the symbol table.


Symbol Tables

Tables: TestStepLocals, TestStepParms, SequenceLocals, System

Select a symbol table from the list to edit or view it. You may also add or remove links to external symbol tables. (Use File=>New or File=>Open to create and edit external symbol tables).

Name	Value	Type	Attributes	Description
var_1	0	Int32		

How Actions Control Switching

Many tests require that the test system’s hardware be in some known state before the measurement begins. For example, suppose you were testing a UUT whose output depended on a specific waveform at its input. Prior to making the measurement in this test, you probably would need to set up a power supply, a signal source such as a signal generator, and a detector such as a frequency counter. You probably would use one or more setup/cleanup actions in the test to set up these conditions, followed by an execute action to make the measurement.

Besides setting up the hardware, you also need to establish whatever connections are needed between the hardware and the UUT before the measurement begins. These connections are called a “switching path.” For example, the power supply needs to be connected to the UUT's power and ground pins, the signal generator needs to be connected to the UUT's input, and the frequency counter needs to be connected to the UUT's output.

Unless the scope of your testing needs is very limited and can be “hard-wired” permanently, you probably will make these connections via some form of switching module that contains a relay matrix. Thus, your test requires some means of controlling the relay matrix to set up the desired switching path.

If you are using hardware handler software to communicate with your switching hardware (see "About Hardware Handlers"), the Test Executive provides a convenient "switching action" to control switching setup and cleanup. A switching action is a special kind of built-in setup/cleanup action that closes connections, such as switching paths made with relays, at the beginning of a test and controls the status of those connections when the test ends.

Note

Unlike other kinds of actions, you do not use the Action Definition Editor to create switching actions. Instead, switching actions are predefined and you insert them as you create tests.

The following happens when a switching action closes one or more switching paths during a test:

1. All of the connections defined for the switching path are closed.
2. The Test Executive waits for the closures before continuing.
3. The action or actions following the switching action are executed.
4. Depending upon which option you specified for the switching action, at the end of the test the connections can remain in their current state, open, or be restored to their previous state.

Caution

HP TestExec SL has no way of knowing if you are "hot switching"—i.e., switching with power applied—during a test. Thus, it is your responsibility to see that switching actions are done at appropriate times.

Two more ways to control switching from actions are:

- If you are using hardware handler software, you can write actions that make calls to a special Hardware Handler API that controls switching paths.

  This method lets you explicitly modify switching paths during a test, which makes it quite versatile. Because it requires you to write code, however, it is more difficult to use than the Switching Path Editor.

For more information about the Hardware Handler API, see Chapter 3 in the *Reference* book.

- If you are <u>not</u> using hardware handler software, you can write actions containing code that communicates directly with your switching devices—i.e., does not use features in the Test Executive—via whichever I/O strategy you have chosen.

  This method potentially is the most versatile because it lets you do anything allowed by your I/O strategy. However, it also can be the most complex and problematic because it does not let features of the Test Executive assist with the task. For example, your code may have to track the states of relays, ensure that “break before make” rules are enforced, and such.

About Exceptions

What is an Exception?

Insofar as HP TestExec SL's testing environment is concerned, exceptions are errors or unusual events that you would not normally expect to happen during testplan execution. Exceptions can be raised by the underlying code on which the Test Executive is built, or by user-defined routines inside actions.

Note Although they are similar in concept, exceptions in HP TestExec SL are distinct from exceptions in a programming language or in an operating system.

How Does HP TestExec SL Handle Exceptions?

HP TestExec SL handles exceptions like this:

- The Test Executive environment provides a specific "Exception sequence," which is a branch of the testplan that automatically executes when the system detects an exception.

  The Exception sequence should put the test system and UUT in a safe state. For example, the Exception sequence should discharge capacitors on the UUT, reset signal sources, reset power supplies, and reset any remaining instruments. Since the Exception sequence should perform only a safe, no-assumptions shutdown of the system, it may be slower than other operations. You can edit an Exception sequence just as you would any other sequence of tests.

  For an example of using the exception sequence, see "Branching on an Exception."

- If there are no tests in the Exception sequence, the system raises an exception to be caught by the user interface, signaling that the system could not be shut down properly.

- Test groups do not execute after an exception occurs. When an exception occurs, the test may have only partially completed, leaving the hardware in an unknown state. Therefore, a test group’s normal clean-up actions may be invalid. Again, use the Exception sequence to do all clean-up for exception conditions.
- HP TestExec SL treats exceptions that reach the test environment as “abort” conditions and stops executing the test and testplan immediately. The Exception sequence then executes automatically.
- The user interface controls whether testing continues after an exception has occurred. The interface can determine if an exception occurred during test execution. The user interface can also query if any exceptions occurred during the Exception sequence to decide if a safe shutdown has occurred. The user interface decides whether to allow restarting the testplan on the same or a new UUT. The user interface can also decide whether to continue looping on a testplan.
- The user interface treats exceptions differently for developer, operator, and automation interfaces. For exceptions in the developer interface, the user interface displays a dialog box and stops, even if a successful shutdown has occurred. If you write your own operator or automation interface, you may want to continue at the next testplan or module after an error. This lessens the chances of an exception shutting down a production line.

Where Should I Handle Exceptions?

Whenever possible, you should handle exceptions within actions so they never cause branching to the Exception sequence. For example, you may know that a non-fatal error–such as a time-out for an instrument–might occur and want to handle that exception in the action. You can then construct the action routine so it will catch, check, and clear the exception and then continue to fail or retry the test as needed. Other exceptions include such conditions as an action's parameters not matching the parameters defined in the Action Definition Editor, memory allocation errors, and so on.

If you need to completely stop execution from within an action, you can create a dialog box requiring a response from the user. This forces human intervention before testing can continue.

Exception handling in actions is described with language-specific topics about creating actions in the *Using HP TestExec SL* book. Also, see Chapter 4 in the *Reference* book for a description of the functions in the Exception Handling API, which lets you raise and handle user-defined exceptions plus examine and handle exceptions raised in the testing environment.

About Switching Topology

What is Switching?

Most tests require that the test system's hardware be in some known state before the measurement begins. Besides setting up the hardware, such as programming power supplies and instruments to known values or states, you need to establish connections between the hardware and the UUT before making the measurement. Many of these connections are not permanently "hard wired" but are controlled programmatically via some form of "switching." Switching, multiplexing, and signal routing exist because it is not cost effective to have every instrument or other resource behind every pin.

What is Topology?

The Test Executive cannot control switching unless it knows which programmable signal paths exist for your specific test system hardware. We refer to this information collectively as the switching topology, or simply "topology," available for testing a given UUT. Topology information includes definitions of the modules, wires, switches, and buses of the test system that are interconnected by switching.

How Switching & Topology Interact

Assuming that you have hardware handler software (described in "About Hardware Handlers") for each programmable module that does your switching, a software tool called the Switching Topology Editor lets you define or describe your hardware so the system software is aware of its characteristics.[1] This process maps a logical view of your system's hardware onto its physical reality. For example, you can associate a physical point, such as a pin on the UUT, with a logical node name that is easy to use and

1. You cannot use the Test Executive's graphical features, such as the Switching Path Editor, to control switching unless you use hardware handler software.

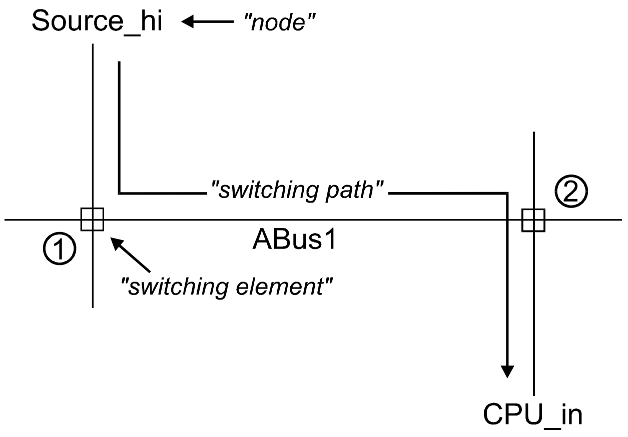
remember, such as `UUT_Signal_in`. From then on, you can refer to the node name, or an alias for it, and the Test Executive knows which physical point you mean. This level of abstraction lets you focus on developing tests instead of trying to remember details of the hardware.

Once you have entered this topology information and saved it, you can use the Switching Path Editor to specify switching actions that tell the Test Executive how to control programmable paths during testing. A switching action sets up connections, such as those made via a relay matrix module, needed when a test begins. It also controls the state of those connections when the test ends.

A Closer Look at Switching Topology

Switching Paths

As shown below, electrical end-to-end connections in a test system are made via a “switching path” that consists of one or more individual connections made by “switching elements” that interconnect “nodes.”



Here, the switching path connects three nodes whose names are `Source_hi, ABus1,` and `CPU_in`. `Source_hi` might be some form of stimulus, and `CPU_in` might be a pin on the UUT. They are connected by closing two switching elements, denoted 1 and 2, located on a bus named `ABus1`. The switching elements themselves might be relays on a switching card.

If we wanted to describe the switching path above in a single, readable statement it might look like this:[1]

[Source_hi ABus1 CPU_in]

This means that `Source_hi` connects to `ABus1`, which connects to `CPU_in`. Note that `Source_hi`, `ABus1`, and `CPU_in` are logical names—i.e., labels—that are convenient to use but do not actually describe where the physical node is located. Because there is a single switching element connecting them, we say that `Source_hi` and `ABus1` are “adjacent.” In a similar fashion, `ABus1` and `CPU_in` also are adjacent.

Having seen the above, we now know that:

- A node is any electrically common, uninterruptable point in the topology.
- Each node has a name, or label.[2]
- Two nodes are adjacent if there is only one switching element connecting them.
- A switching path is an ordered set of adjacent node names.

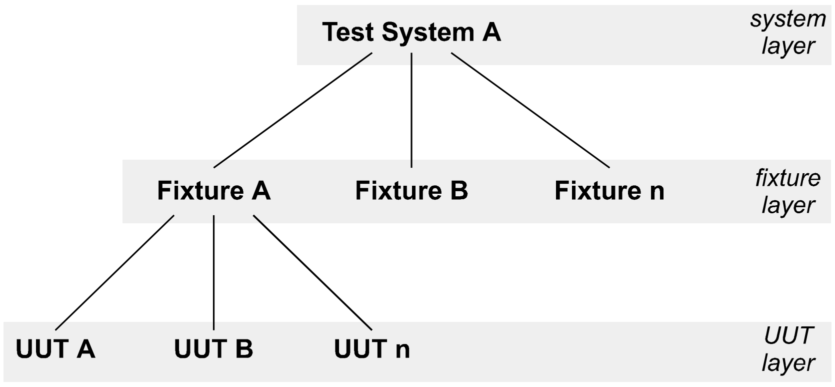
Why not simply call a switching element a relay? Calling it a relay is an oversimplification because other kinds of switching elements exist. For example, multiplexers and rotary switches also are switching elements, except they have multiple positions while a relay has only two: open and closed.

Switching becomes more complicated when all the elements that form a switching path are not located on the same card or module. Then, the switching path has to include a specifier to identify which switching element on which card is being used to make the connection.

1. This is the notation that appears in the Switching Path Editor when you use it to define switching paths.
2. Nodes also can have multiple names, or aliases, for convenience in referencing them.

The Three-Layer Model for Switching Topology

HP TestExec SL uses a three-layer model to define a test system's switching topology. As shown below, the first layer defines the system hardware, the second defines one or more removable fixtures used with the system hardware, and the third defines one or more UUTs used with a given fixture.



Information typically defined at the system layer includes:

- Definitions for any cards or modules used in the system, including adjacent switching elements.
- A definition of the cabling that connects the cards or modules.
- Definitions of aliases for system resources.

Information typically defined at the fixture layer includes:

- Definitions of wires in the fixture.
- Definitions for the names of any edge connectors.
- Definitions for any electronics inside the fixture that is a part of your switching strategy.

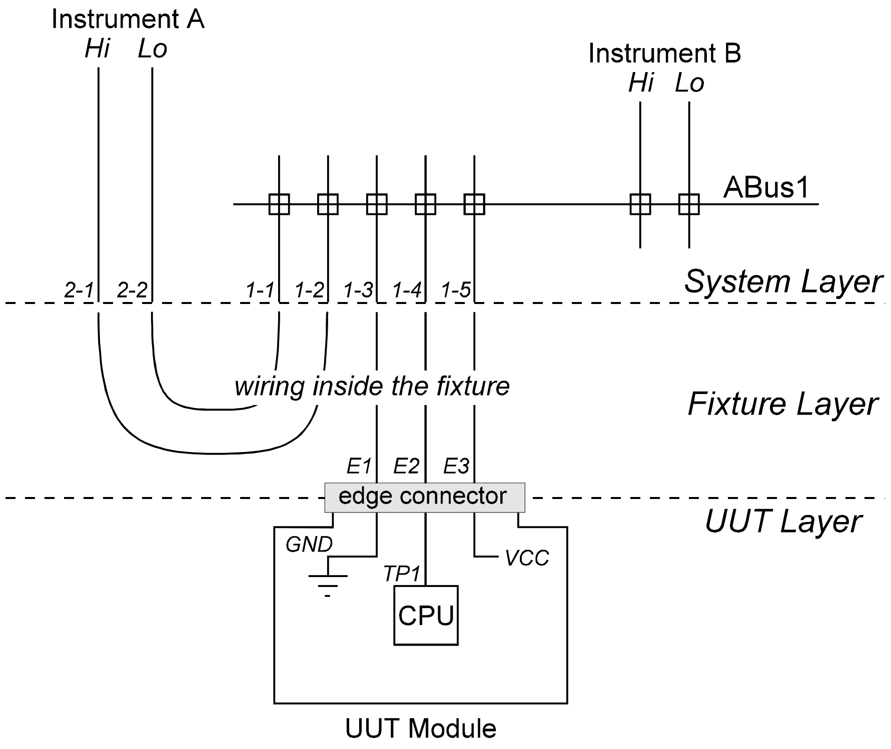
Information typically defined at the UUT layer includes:

- Definitions of aliases for test points on the UUT.

Each layer in the switching topology lets you define:

aliases	Aliases are convenient, alternate names for node names. For example, in the system layer you might call a system resource `DVM_high` instead of `MCM:Inst1`. Or, in the UUT layer you might call a test point on the `UUT CPU_in` instead of `Edge_Connector_Pin_2`. Besides improving the readability of names, aliases increase the portability of tests and testplans across test systems. Each node can have one or more aliases.
wires	Wires include wires, cables, and jumpers. In the system layer, these wires could describe how you have cabled together the test system's cards or modules. Or, in the fixture layer you could describe interface pins that have been shorted together.
modules	Modules are "boxes," cards, or groups of programmable switches that need to be managed by the switching software.

What do these actually mean? Refer to the next example, which shows how the conceptual layers might relate to actual hardware.



Although it may seem complex at first, having three separate layers of switching topology increases the likelihood that you can reuse individual layers. For example, you only have to modify the UUT Layer when using an existing fixture and test system to test a new UUT. This can be very convenient if you have a family of modules to test that can share common resources and fixturing but whose internal details vary.

About Hardware Handlers

Hardware Handlers in General

A hardware “handler” is an additional layer of software between HP TestExec SL and a device driver. By providing a set of standard—i.e., “well-known”—functions through which HP TestExec SL communicates with device drivers, a handler enhances HP TestExec SL's ability to control devices.

The best way to understand what a hardware handler does is to understand what happens if a handler is not used. When using a conventional driver strategy, the actions in a test talk to instruments, switching modules, or other devices via a device driver as the test executes. This means that each action must communicate via a specific control language or set of commands understood by the device driver. Thus, actions are specific and unique insofar as an action that controls an instrument via one control language cannot readily be reused to control other instruments requiring a different language.

But when a handler is used, actions can communicate with devices via a standard set of function calls known to HP TestExec SL. The handler translates these standard calls made by actions into the specific control language needed to communicate with a driver or instrument. Because the function calls generally remain the same from device to device—the exception being specific functions that one device supports but another does not—it is much easier to reuse actions (and tests and testplans built from them) across various kinds of test systems.

Switching Handlers in Particular

A common type of hardware handler is a “switching handler,” which contains routines that know how to communicate with a switching module and are aware of its topology. You do not call switching handlers directly when using them. Instead, you use a “switching action” in your tests to call the switching handler for you. When the switching handler is called, software that manages switching decides which function to call based on the high-level (symbolic) names in the paths, the topology of the system hardware, the fixture, and the names of UUTs.

What's Inside a Hardware Handler?

A hardware handler contains code, written in C, that implements the functions called by HP TestExec SL when it interacts with hardware. A few of the functions are general-purpose enough to be useful with various kinds of hardware modules. These functions can:

- Open (initialize) a module.
- Close a module.
- Reset a module to a default state (which can be different from its initialized state).
- Declare any parameters needed to create a unique instance of the handler, such as which instrument identifier to use.

A hardware handler also can contain specialized functions that are used to interact with switching hardware; i.e., when the hardware handler is used as a switching handler. These functions can:

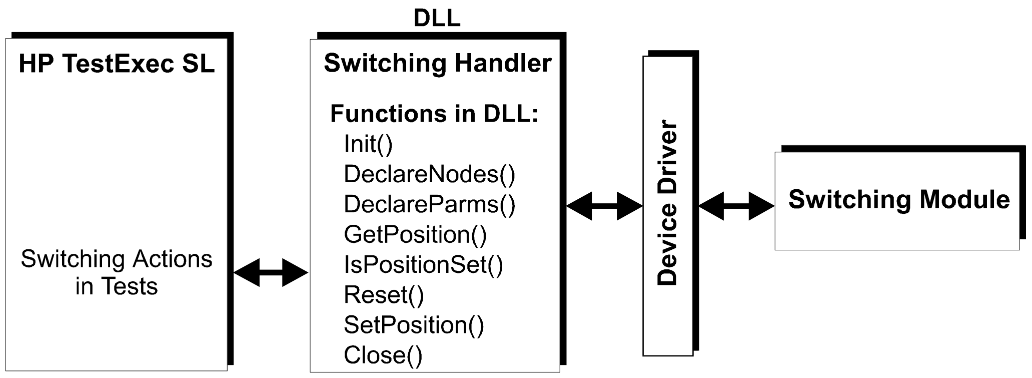
- Set the position of an element in a switching module.[1]
- Return the current position of an element in the switching module.
- Declare nodes that define the topology of the switching module and the switching elements that make them adjacent.
- Optionally improve the speed of switching by letting switching elements open and close in parallel with one another instead of sequentially.

Each hardware handler resides in its own DLL. You need one hardware handler for each type of module you wish to control from HP TestExec SL.

1. An "element" is a programmable connection, such as a relay or a multiplexer.

However, a single hardware handler can control more than one module because it is passed one or more parameters that specify a unique instance of the module to control.

The example below shows a hardware handler used to control switching hardware, which means it is a switching handler. A switching handler contains functions called by switching actions. The various functions inside the DLL, some of which include calls to the switching API, control the operation of the switching module. You do not need to write code that calls these functions. Instead, your task in creating a switching handler is simply to make the functions do whatever you wish them to do when they are called by HP TestExec SL.



The functions in a switching handler let HP TestExec SL communicate with the switching module via the switching handler. This communication is two-way insofar as HP TestExec SL can request status information, such as the current position of a switching element, as well as control the operation of the switching module.

Caution

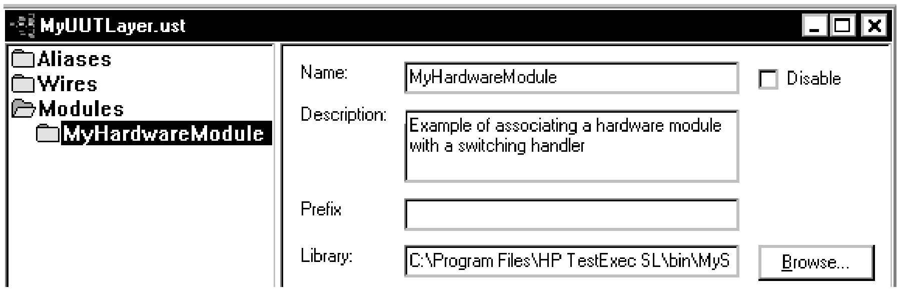
Do not use both a switching handler and a direct I/O strategy to control the same switching module. Because a switching handler tracks the states of the switching elements in a switching module, if you directly manipulate the switching module the handler will not be aware of it and may assume the wrong state.

How Do Switching Actions Use Switching Handlers?

As stated earlier, a switching handler’s purpose is to let switching actions in tests in your testplan control switching hardware. However, the way in which switching actions, layers in the switching topology, switching handlers, and switching hardware work together may initially seem obscure because various software tools are used to create the individual components.

The relationships among switching actions, switching topology layers, switching handlers, and switching hardware are established by the steps below.

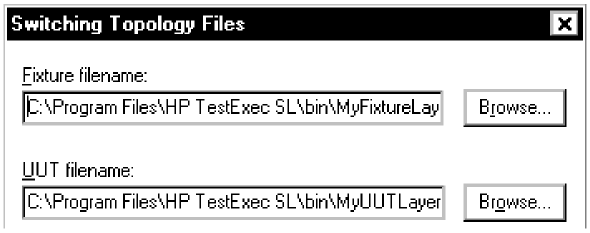
- You use the Switching Topology Editor to define topology layers and associate switching handlers with them. This establishes the relationship between physical hardware resources and logical resources defined in the topology layers, such as switchable nodes and wires.



Name: MyHardwareModule
Description: Example of associating a hardware module with a switching handler
Prefix
Library: C:\Program Files\HP TestExec SL\bin\MyS

When defining topology layers, you specify one switching handler per type of hardware module. When there are multiple modules of the same type, you can pass parameters (such as a VXIbus address) to the switching handler to create unique instances of the hardware modules.

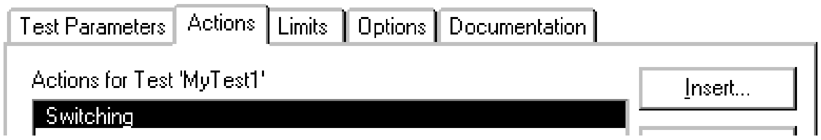
- After creating a testplan, you specify which switching topology files to use with the testplan. This makes switchable paths defined in the switching handler visible to your testplan.



Fixture filename: C:\Program Files\HP TestExec SL\bin\MyFixtureLay

UUT filename: C:\Program Files\HP TestExec SL\bin\MyUUTLayer

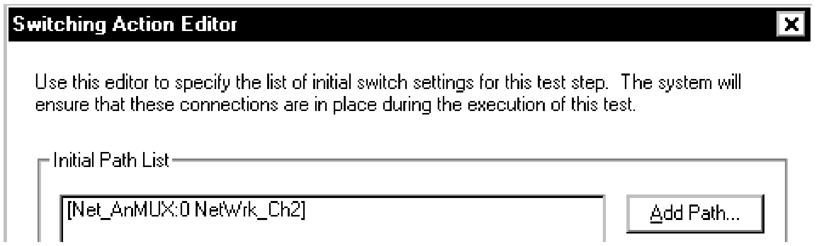
- You use the Test Executive’s features to create switching actions in tests in your testplan.



Actions for Test 'MyTest1'

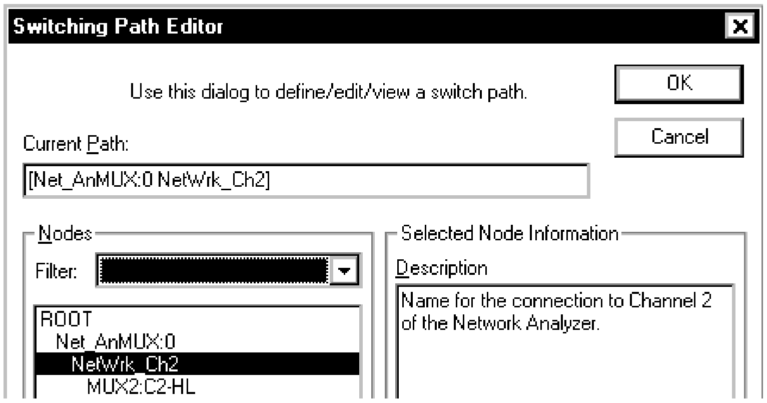
Switching

- You use the Switching Action Editor and Switching Path Editor to specify what each switching action should do.



Use this editor to specify the list of initial switch settings for this test step. The system will ensure that these connections are in place during the execution of this test.

[Net_AnMUX:0 NetWrk_Ch2]



Use this dialog to define/edit/view a switch path.

Current Path: [Net_AnMUX:0 NetWrk_Ch2]

```
ROOT
  Net_AnMUX:0
    NetWrk_Ch2
      MUX2:C2-HL
```

Description: Name for the connection to Channel 2 of the Network Analyzer.

After you have done the preceding, the switching paths specified in switching actions cause HP TestExec SL to make calls to the switching handler, which in turn makes calls to a device driver or other means of controlling the switching hardware.

Where Do I Get a Hardware Handler?

Custom test systems sold by Hewlett-Packard that use HP TestExec SL may come with prewritten hardware handlers appropriate for their specific hardware. Otherwise, you must write your own hardware handler, as described in Chapter 2 of the *Customizing HP TestExec SL* book.

About Test & Action Libraries

Libraries in General

In HP TestExec SL a library is a directory containing related items that are potentially reusable. The types of libraries supported by the Test Executive are:

Test Library	Contains test definitions that provide the structure for tests. You can save any test in a testplan to a test library.
Action Library	Contains the actions from which tests are built.

You can search for library entries by name, add new entries, and modify or delete existing entries.

Test Libraries

Reusing existing tests can greatly reduce the amount of work needed to develop testplans for future UUTs. Although not every test that you develop will be a potential candidate for reuse, you probably will create some that are important enough to reuse as-is or as templates for new, similar tests.

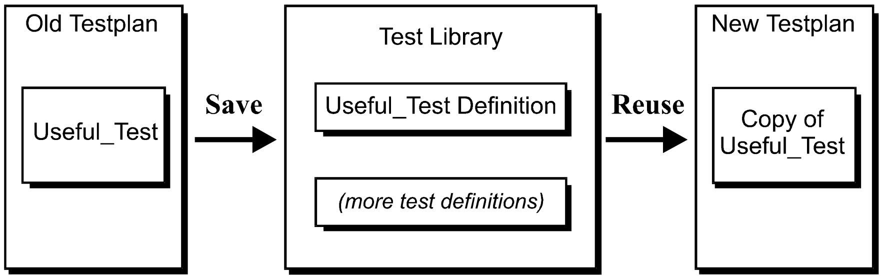
During the test development process you can save a test definition, which is a copy of a test suitable for use as a template, in a test library. Each test definition contains:

- A list of actions used in the test.
- The parameters for the actions, plus default values for the parameters.[1]
- A list of results associated with the test.

1. Only one set of values for parameters and limits—i.e., the values for one named variant—is stored with each test definition.

- Default values used for limits checking by comparing the actual results against the desired results.

As shown below, you reuse an existing test definition by inserting a copy of it into a new testplan. In many cases, you will need to modify the existing definition's parameters to fit the circumstances of the new test. For example, you may need to specify which UUT pin is to be tested and modify the limits for the reused test.



Test libraries are organized in a directory structure you can customize to meet your needs. For example, you may want a directory for general-purpose tests as well as individual directories that contain tests for particular types of UUTs.

The name of each test library is the same as that of its directory. Inside the test library directory, each test definition has a unique name, which is the name of the test followed by the extension “utd”. For example, you might have a test named “ileak” defined in file “ileak.utd” in directory “my_tests”. Thus, the name of the test library is “my_tests”.

Action Libraries

Action libraries contain the actions used to build tests. They are most useful when they store actions that do a single task because simple actions potentially have greater reusability than more complex actions. Although some actions are provided with the Test Executive, you are likely to create many more to address your specific testing needs. Thus, action libraries tend to become customer-specific over time.

Actions consist of a definition or source and a DLL that contains the action's executable code. The definition contains:

- The name of the DLL in which the action code is found.
- A description of the action.
- The type of action, which can be C parameter block, National Instruments LabView, or HP VEE calling sequence.
- Definitions of parameters used in the action, including the parameter's type, default value, and description.

Note Action names should be unique so the Test Executive can identify each one.

The definition for each action resides in a file whose extension is “umd”; e.g., “dmmsetup.umd”. Action libraries are organized in a directory structure you can customize to meet your needs. Each action has two files associated with it: its definition, and its DLL. The DLL need not be in the same directory as the definition.

We recommend that you organize action libraries under a root directory, use the root directory to hold all action DLLs, and define as many subdirectories (libraries) as needed to hold the action definition files.

Development Versus Production Libraries

When developing code, it is a common practice to store the code in a special, private library until it is stable enough for use in a production environment or by other developers. And after the code has been released to production, there may be times when you need to enforce changes to testplans. For this reason, the organization of libraries used for development need not be the same as the organization used in a production environment.

If you need to move individual library entries, or groups of them, follow these guidelines:

- Be sure that the library search path (tests, actions, and the DLL search path list) on any given system reference directories containing the tests, actions, and handler routines needed by any testplan to be run on that system.

- If you move the action or instrument handler library entries, be sure to move both the definition file and the DLL.

- Be sure the names of library entries are unique so the correct one will be found and used.

What Belongs in a Library?

The previous topics described the characteristics of libraries, and suggested how to use them, but did not suggest what belongs in libraries. Ideally, the tests and actions that you store in libraries should have the following characteristics:

- They should be appropriate for reuse.

  It is in your own best interests to keep reusability in mind when developing tests and actions. Generally speaking, simple, general-purpose tests and actions tend to be more reusable than larger, more specialized pieces of code. However, complexity is not always the determining factor.

  For example, suppose you were designing a test for a module that was one in a family of similar modules. If you made the test just specific enough to test whichever features all the modules in the family had in common, you probably could reuse the test on all the modules simply by providing it with new parameters and limits. Tests and actions that exhibit good reusability are comprehensive but not so large that they are cumbersome.

- They should use logical names for pins on the UUT, not specific pin identifiers or other “hard-coded” information.

- They should not depend upon a specific sequence of actions or tests preceding them for correct operation.

Note

If you save in a library a test whose setup or cleanup tasks are derived from a surrounding test group, the test saved in the library loses those setup or cleanup tasks. Thus, you should indicate this dependency in the descriptive comments for the test to make others aware of this potential problem.

Glossary of Terms

This glossary provides definitions of terminology that may be unfamiliar or unique to HP TestExec SL. The definitions are in alphabetical order.

Action

The smallest component of a test or test group. Most actions have a name and call a user-written routine that does something useful, such as make a measurement. The types of actions are execute, setup/cleanup, and switching.

Action Definition Editor

A software tool used to create actions, which are the building blocks used to create tests and test groups.

Action routine

Executable code, written in a standard programming language, that is called by an action. An action routine does something useful, such as making a measurement.

Action style

An action's “style” determines the method in which parameters are passed to its action routines. The style depends upon which programming language is used to write the action routines.

Actions can have the following styles:

DLL	(Written in C/C++) Parameters are passed in a named group or “block” of parameters. This style is highly recommended as the fastest, easiest to create, and easiest to maintain.
HP VEE	(Written in HP VEE) Parameters are passed in a named block or group of parameters to be graphically “wired” to an HP VEE function.
LabVIEW	(Written in National Instruments LabVIEW) Parameters are passed in a named block or group of parameters to be graphically “wired” to a National Instruments LabVIEW virtual instrument (VI).
HP RMB	(Written in HP BASIC for Windows) Parameters are passed via a list of parameters in a server program.

Adjacency

Two nodes in the switching topology that can be connected by a switching element.

Alias

An alternate name for an item in the switching topology. Aliases let you use convenient names when defining topology; for example, you could assign node “MCM:Inst11” an alias that is easier to remember, such as “ScopeInput”. Each node can have one or more aliases.

Note that you cannot use aliases for the names of modules.

API (application programming interface)

In the context of HP TestExec SL, a large number of predefined functions provided for use in the code that you write for a test system. HP TestExec SL includes the:

- C Action Development API, which provides functions that let you use a C/C++ compiler to develop action routines.
- Exception Handling API, which provides functions that let you raise and examine exceptions that occur during testing. Also, it includes functions that let you programmatically abort testing if an exception occurs.
- Runtime API, which provides functions that let you replace the default user interface for operators with a custom interface.
- Hardware Handler API, which contains functions used when writing a hardware handler.

Datalogging

The process of collecting data about tests when the testplan runs. Subsequent study of this data can aid you in improving the processes associated with manufacturing and testing.

Data container

A method of encapsulating data beneath a layer of abstraction that hides the data's complexity and increases its portability across programming environments. A data container is an object.

DLL (dynamic-link library)

A library of software code that is automatically loaded and unloaded as needed.

Exception

An error or unusual event that you would not normally expect to happen during testplan execution. Exceptions can be raised by the underlying code on which the Test Executive is built, or by user-defined routines inside actions.

Note Although they are similar in concept, exceptions in HP TestExec SL are distinct from exceptions in a programming language or operating system.

Execute action

A type of action typically used to make a measurement. Each execute action has a name and calls user-written code that does a task. You use the Action Definition Editor to define an execute action and a programming environment, such as Visual C++, to implement its code.

Hardware handler

A software layer between HP TestExec SL and the driver for a hardware module. By providing a set of standard—i.e., “well-known”—functions through which HP TestExec SL communicates with device drivers, a handler enhances HP TestExec SL's ability to control devices.

Keyword

An identifier used to restrict the number of matches found when searching for a specific item. Keywords often describe the item; for example, suitable keywords for an action might be “trigger” or “range” to identify what an action does or how it is used.

Library

A collection of related code stored in one or more directories. HP TestExec SL supports libraries of actions and libraries of tests. Organizing actions and tests into libraries makes it easier to find and manage existing code so you can reuse it.

Limits checker

A feature that decides whether a test passed or failed by comparing the test’s results against predefined criteria. The kinds of limits checkers provided by HP TestExec SL include:

Min/Max	The test passes if its result is within specified minimum/maximum values. To pass, the result must be greater than or equal to the minimum and less than or equal to the maximum.
Equivalence	The test passes if its result exactly matches a specified value.
Nominal Tolerance	The test passes if its result is within a specified +/- tolerance of a specified nominal value. To pass, the result must be greater than or equal to the lower tolerance value and less than or equal to the upper tolerance value. *Note:* The nominal tolerance is a simple number and not a percentage. In other words, a nominal tolerance of 5 means the result must be within plus or minus 5 of the nominal value, not within plus or minus 5 percent of it.
<No Limits>	The test is not checked against pass/fail limits.

Master keyword

A keyword stored on a predefined list in the Action Definition Editor’s initialization file. You can use the Action Definition Editor to add a master keyword to any action.

Module

A hardware resource in the switching topology, such as a VXIbus instrument.

Node

Any electrical point in the switching topology. Each node has a name, or label.

Operator interface

A user interface whose main purpose is to let system operators interact manually with HP TestExec SL. For example, a typical operator interface might have onscreen buttons labeled "Start" and "Stop," as well as status indicators labeled "Pass" and "Fail" to report the results from testing. System operators might use a mouse to "press" these buttons and then accept or reject UUTs based on their pass/fail status after the testplan has run.

Parameter block

A list of named parameters stored in a uniquely named collection or “block.” When you need to use the parameters, you specify a handle to the parameter block instead of specifying the full list of parameters. Parameters in a parameter block are looked up by name and not by their position in the block.

Profiler

A software tool you can use to see how long each action or test group in a testplan takes to execute. Once you know how long each action or test group takes to execute, you can decide where to begin the "tuning" process, and monitor any improvements you make.

After enabling the profiler, you run a testplan to collect data, and then either view Pareto charts directly in HP TestExec SL or use a financial spreadsheet program to further analyze the data.

Routine type

The identifier of how an action routine is used. You can specify either setup, cleanup, paired setup/cleanup, or execute routines, where:

- Execute routines contain a single entry point and typically do a single task, such as making a measurement.
- Setup routines contain a single entry point and typically do a single task, such as setting up a power supply before making a measurement.
- Cleanup routines contain a single entry point and typically do a single task, such as resetting a power supply after making a measurement.

- Paired setup/cleanup routines contain two entry points and typically bracket (surround) one or more execute actions. For example, the setup component of a setup/cleanup routine could set the UUT to a particular mode, an execute routine could make a measurement, and the cleanup component of a setup/cleanup routine could return the UUT to its idle mode.

Sequence
A named series of test groups, tests, and flow control statements executed in a predefined order.

Setup/cleanup action
A type of action typically used do tasks before and after execute actions. Each setup/cleanup action has a name and calls user-written code that does a task. You use the Action Definition Editor to define a setup/cleanup action and a programming environment, such as Visual C++, to implement its code.

Switching action
A type of action that sets up connections, such as switching paths made with relays, at the beginning of a test and controls the status of those connections when the test ends. Unlike other types of actions, you do not use the Action Definition Editor to create switching actions; instead, they are built into HP TestExec SL. Also, switching actions do not have individual names.

Switching element
A programmable connection, such as a relay, between two nodes in a switching path.

Switching handler
A common type of hardware handler. When you use a switching handler with a switching module, you can use the Switching Topology Editor to define your test system's topology and then use the Switching Path Editor to conveniently control switching paths during a test or test group.

Switching path
A necessary connection between nodes during a test or test group. The connection is made via one or more switching elements, such as relays.

In the example of a switching path below, the source bus of the function generator is connected to the measurement bus of the oscilloscope when the switching path is closed.

[FuncGen SrcBus Scope MeasBus]

Switching state

A “snapshot” or stored copy of one or more switching paths and the states of the switching elements in those paths.

Switching topology

A combination of physical and logical descriptions that define the switching configuration and interconnections between resources and the UUT, which includes definitions for the modules, wires, switches, and buses of the test system. These definitions map a logical view of your system's hardware onto its physical reality, and add a level of abstraction.

Switching Topology Editor

A software tool used to define switching topology and to make the Test Executive aware of hardware modules that are available as resources during testing.

Switching topology layer

Switching topology is defined in three layers: system, fixture, and UUT. The first layer defines the system hardware, the second defines one or more fixtures used with the system hardware, and the third defines one or more UUTs used with a given fixture.

Information defined at the system layer includes:

- Definitions for any cards or modules used in the system.
- A definition of the cabling that connects the cards or modules.
- Definitions of convenient names—i.e., aliases—for system resources, such as “DVM_high”.

Information defined at the fixture layer includes:

- Definitions of wires in the fixture.

- Definitions for the names of any edge connectors.
- Definitions for any electronics inside the fixture that is a part of your switching strategy.

Information defined at the UUT layer includes:

- Definitions of convenient aliases for test points on the UUT, such as “TP1.”

Symbol table

An unordered, named collection of data items (variables) called “symbols” whose usage has a specific scope. For example:

- The symbol table named System contains symbols associated with the testing environment, such as the user ID, test system ID, and serial number of the UUT. Its scope is the testplan.
- User-named external symbol tables are supported. Their scope is the testplan with which they are associated.
- The symbol table named SequenceLocals contains symbols whose scope is a sequence. Variables defined in it can be used to pass values between tests or test groups because the variables are visible within a given sequence throughout the testplan.
- The symbol table named TestStepLocals contains symbols whose scope is a specific test or test group. Variables defined in it can be used to pass values between actions inside the current test but not to actions in other tests or test groups.
- The symbol table named TestStepParms contains symbols whose scope is a specific test or test group. Variables defined in it contain parameters passed to the test or test group.
- The symbol table named TestPlanGlobals contains symbols whose scope is global to the testplan and all tests and actions in all sequences. Variables defined here can pass values anywhere within a testplan.

Test

A named series of actions executed as a group. A test can contain execute, setup/cleanup, and switching actions.

To be meaningful, most tests use a limits checking feature that determines if the unit under test passed or failed the test. Also, most tests use a datalogging feature to store information collected during the test.

Test Executive

A software tool used to develop tests, assemble them into a testplan, run the testplan, and evaluate the pass/fail results.

Test group

An optional, named set of tests executed in a predefined order. Each test group can have an associated list of setup/cleanup actions that do setup and cleanup tasks for all the tests bounded by the test group, and a list of switching actions. A test group is bounded by “testgroup <name>” and “end testgroup” statements inside a testplan. Test groups can be nested inside test groups.

Test limits

The acceptable boundaries for a test. For example, if the results from a test are less than the lower limit or greater than the upper limit, the test fails. A test can have more than one set of limits, where each set is associated with a named variant, such as “Hot” or “Cold.”

Test procedure

A group of measurement routines that comprise a test; i.e., another name for a test.

Testplan

A named sequence of tests executed in a predefined order to test a specific device or unit under test. A testplan also can be further divided into groups of tests called “test groups.”

UUT (unit under test)

The unit, module, device, etc. being tested. Sometimes referred to as a DUT, or device under test.

Variant

A mechanism that lets you specify which named variation on a testplan is executed when you run the testplan. Because the let you use *one* testplan with *n* different sets of test limits and parameters, variants are useful where one UUT is very similar to another except for slightly different values for its test limits or parameters.

Each variant lets you:

- Use the same sequence of tests with different parameters and limits.

  For example, you may want to specify different limits for various temperatures at which the tests are executed.

- Control which set of tests is executed for a given testplan.

  For example, the set of tests used by Quality Control may be a superset of the tests used by Production.

- Change the testing algorithm as desired.

  For example, a testing algorithm used by Quality Control may need greater precision than a testing algorithm used by Production.

The name of the default variant is Normal. Other typical variants might be named Hot or Cold.

VXIplug&play

An industry standard that lets you program standalone and VXIbus instruments using various programming languages, such as HP VEE, Visual Basic, and Visual C++. VXI*plug&play* drivers have a consistent architecture, and are developed and used in a consistent fashion. They let vendors of instruments develop drivers for their own instruments, and ensure that those drivers are interoperable with drivers provided by other vendors.

Wire

A bus or other connection in the switching topology.

Index