HP TestExec SL

Using HP TestExec SL

Notice

The information contained in this document is subject to change without notice. Hewlett-Packard Company (HP) shall not be liable for any errors contained in this document. HP makes no warranties of any kind with regard to this document, whether express or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Warranty Information

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19 (c) (1,2).

Use of this manual and magnetic media supplied for this product are restricted. Additional copies of the software can be made for security and backup purposes only. Resale of the software in its present form or with alterations is expressly prohibited.

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.

E1074-90008 — Software Rev. 1.51 — Rev. B - June, 1996

E2011-90012 — Software Rev. 2.00 — Rev. C - January, 1997

E2011-90015 — Software Rev. 2.10 — Rev. D - May, 1997

E2011-90019 — Software Rev. 3.00 — Rev. E - January, 1998

E2011-90023 — Software Rev. 4.00 — Rev. F - August, 1999

About This Manual

This manual describes how to do tasks of interest to most users of HP TestExec SL.

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.

Items you must specify are italicized and enclosed by angle brackets, like this:

<*filename.txt*>

which you might replace by typing:

```
MyFile.txt
```

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.

Most 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.

Contents

1. Working With Testplans

3. Working With Actions

4. Working with Switching Topology

6. System Administration

7. Working with VXIplug&play Drivers

8. Using String Formatting

9. Using Actions to Control Message-Based Instruments

10. Testing Multiple UUTs

Index

1

Working With Testplans

This chapter describes how to use testplans, which are named sequences of tests that are executed as a group to test a specific device or unit under test.

For an overview of testplans, see Chapter 3 in the *Getting Started* book.

A Suggested Process for Creating a Testplan

Although we have no way of knowing about your specific hardware, we recommend that you consider the following process when creating a testplan.

Preparing to Write the Testplan

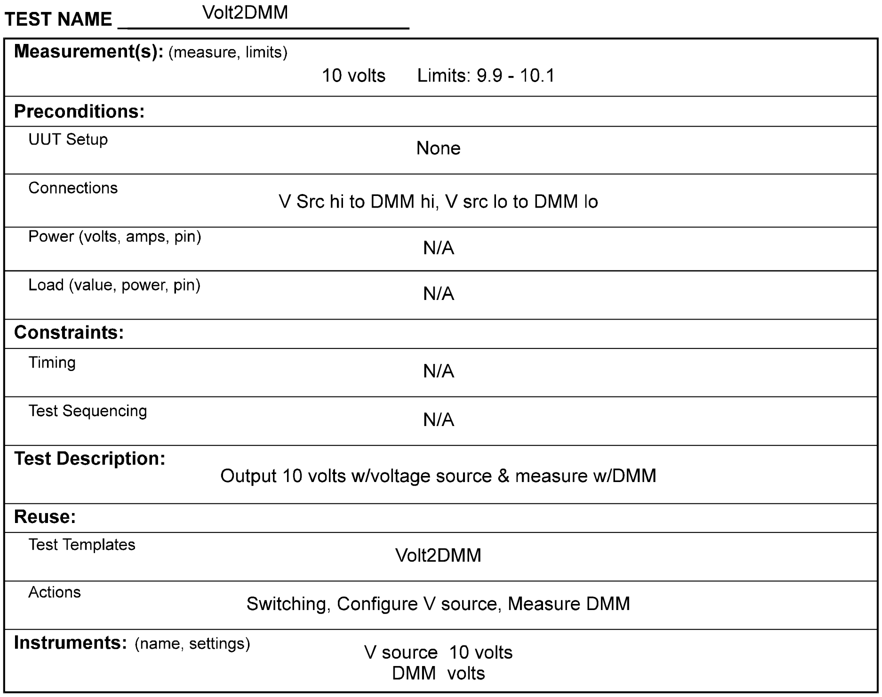
1. Gather the testing specifications and requirements for the UUT (unit under test).

   You must thoroughly understand the UUT before you can test it effectively. This includes both the physical (such as pinouts) and electrical characteristics of the device.

2. Plan the tests and the sequence in which they will be executed.

   Determine which kinds of tests are needed in your testplan (including tests for failure and exception handling, if desired). Determine the order in which the tests should be executed. Given the above, determine where to use test groups.

   Tip: You may find it useful to draw a worksheet and make copies of it to write on when planning tests. For example, the worksheet might briefly describe the test, list the hardware resources needed, the test limits, any setup or cleanup requirements, timing constraints, a list of input and output pins, and such. An example of a typical worksheet is shown below.



TEST NAME Volt2DMM

Measurement(s): (measure, limits)	10 volts Limits: 9.9 - 10.1
Preconditions:	
UUT Setup	None
Connections	V Src hi to DMM hi, V src lo to DMM lo
Power (volts, amps, pin)	N/A
Load (value, power, pin)	N/A
Constraints:	
Timing	N/A
Test Sequencing	N/A
Test Description:	Output 10 volts w/voltage source & measure w/DMM
Reuse:	
Test Templates	Volt2DMM
Actions	Switching, Configure V source, Measure DMM
Instruments: (name, settings)	V source 10 volts DMM volts

3. Plan the system resources for each pin on the UUT.

   Using the information from the previous step, be sure your test system has the hardware resources needed to do the tests. For example, do you have enough power supplies, signal sources, and signal detectors? If not, you must add hardware or find a way to simplify the tests.

4. Plan and build the fixture or other means of connecting the test system’s hardware with the UUT.

   Pins on the UUT must be connected to the test system’s power supplies, signal sources, and signal detectors. If you test various kinds of UUTs on a single test system, you may want to use an interchangeable fixture to make the connections. Or, you need some type of cabling to make the necessary connections.

If you are using programmable switches, such as switching cards, to make connections between resources and the UUT and you have hardware handler software for those switches, you probably will want to use the Switching Topology Editor to define your topology so you can use switching actions in your tests.

Writing the Testplan

1. Add tests and test groups to your testplan.

2. Copy and customize existing tests from libraries where possible. Where needed, add the tests to test groups. If there is no existing test to reuse, create new tests from existing actions in libraries where possible. If no suitable actions exist from which you can create a new test, create new actions, add them to an action library, and then create a new test from them.

3. Tune the tests for performance and reliability.

   This process can be as flexible as you like. For example, you might begin by creating actions, using them to create tests, and then using the tests to create a testplan. But if it is more convenient—for example, if different people are developing the actions and the testplan—you may want to begin with an empty testplan and then expand it by adding tests as the actions needed to create the tests become available.

   For more information about tuning tests, including how to use HP TestExec SL's built-in profiler, see "Optimizing the Throughput of Testplans."

To Create a Testplan

Use the Test Executive’s graphical tools to create a testplan.

1. Click 🗋 in the toolbar or choose File | New in the menu bar.

2. Choose Testplan as the type of document.

3. Choose the OK button.

4. Add one or more tests or test groups to the list shown in the left pane of the Testplan Editor.

   For information about adding tests and test groups, see “Using Tests & Test Groups in Testplans.“

5. Click 💾 in the toolbar or choose File | Save in the menu bar.

6. Enter a name for the testplan.

7. Choose the Save button.

To Specify Switching Topology Layers for a Testplan

The switching topology information for a specific testplan resides in three files whose extensions are “.ust”. These files contain information about the system, fixture, and UUT layers of switching topology. Given that one test system can use many testplans, you must specify which switching topology files to use for a given testplan.

Each test system has one system layer defined for it. The name and location of the file containing the system layer resides in HP TestExec SL’s initialization file. This is described under “System Setup” in Chapter 6.

Although you can locate the remaining two files, which contain the fixture and UUT layers, wherever you like, it usually makes sense to put them with other files used with the testplan. Then you must associate these two topology files with the testplan.

Do the following to associate the files for the fixture and UUT layers with the testplan:

1. Load the testplan.
2. Choose Options | Switching Topology Files in the menu bar.
3. Specify the locations of the files for the fixture and UUT layers.

For an overview of switching topology, see “About Switching Topology” in Chapter 3 of the *Getting Started* book. For detailed information, see Chapter 4 in this book.

Using Tests & Test Groups in Testplans

Note

The Testplan Editor window supports the various mechanisms that Microsoft Windows provides to select multiple items; i.e., holding the Ctrl key as you click multiple items; pressing and holding the mouse's left button and then dragging across multiple items; and clicking the first item in a desired list, simultaneously pressing and holding the Ctrl and Shift keys, and clicking the last item in the list. This means that many of the tasks described for individual tests or test groups also can apply to multiple tests or test groups. For example, if you select multiple tests or test groups, you can copy or delete them as you would a single test or test group.

To Add a New Test/Test Group

1. Click the desired insertion point in a testplan shown in the left pane of the Testplan Editor window.

   The test or test group will be inserted immediately before the line selected as the insertion point.

2. Do one of the following:

   - *To insert a test,* click in the toolbar or choose Insert | Test in the menu bar.

     - or -

   - *To insert a test group,* click in the toolbar or choose Insert | Test Group in the menu bar.

3. Do the following in the right pane of the Testplan Editor window:

   a. Specify a name for the test or test group.

If you are using datalogging, be aware of the following restrictions on the names of tests or test groups:

- If your log data is processed by HP Pushbutton Q-STATS, you must not use slashes (/ or \) in test names.
- If your log data is processed by Q-STATS II, only the first forty letters of the test name are significant.

b. Add any desired actions to the test or test group.

See “To Add an Action to a Test/Test Group” in Chapter 2 for more information.

c. If you wish to use variants to provide multiple versions of the parameters and limits, specify them.

See “To Add a Variant to a Testplan” for more information.

To Add an Existing Test

The easiest way to create a test is to reuse a similar test from a test library.

Note

Be sure the search paths for test libraries are set up correctly or you may not be able to find the test you want; see “Specifying the Search Path for Libraries” in Chapter 5.

1. With a testplan loaded, choose an insertion point in the left pane of the Testplan Editor window.

   The test will be inserted immediately before the line selected as the insertion point.

2. Click ▦ in the toolbar or choose Insert | Saved Test in the menu bar.

3. When the Test Libraries box appears, use it to find an existing test similar to the one you need.

For more information about using the Test Libraries box’s search features, see “Searching for Items in a Library” in Chapter 5.

4. Make a copy of the test under a new, unique name.

5. Modify the existing actions as needed.

   For more information, see “To Specify Parameters for Actions in a Test/Test Group” and “To Specify Limits for Actions in a Test/Test Group” in Chapter 2.

6. Modify the existing parameters as needed.

   For more information, see “Specifying Parameters for a Test/Test Group” in Chapter 2.

To Examine or Modify a Test/Test Group

1. Click a test or test group shown in the left pane of the Testplan Editor window.

2. Use the right pane of the Testplan Editor window to examine or modify the contents of the test or test group.

See Chapter 2 for information about specifying the contents of tests and test groups.

To Move a Test/Test Group Within a Testplan

1. Select one or more tests or test groups in the left pane of the Testplan Editor window.

2. Choose ✂ in the toolbar or Edit | Cut in the menu bar.

3. Click the desired new location for the test or test group.

   If you click an existing line, the test or test group will be inserted before that line.

4. Choose 📋 in the toolbar or Edit | Paste in the menu bar.

To Move a Test/Test Group Across Testplans

1. Run two copies of HP TestExec SL and load the source testplan in one and the destination testplan in the other.

2. Select one or more tests or test groups in the left pane of the Testplan Editor window in the copy of HP TestExec SL that has the source testplan loaded.

3. Choose ✂ in the toolbar or Edit | Cut in the menu bar of the copy of HP TestExec SL that has the source testplan loaded.

4. Click the desired new location for the test or test group in the left pane of the Testplan Editor window in copy of HP TestExec SL that has the destination testplan loaded.

   If you click an existing line, the test or test group will be inserted before that line.

5. Choose 📋 in the toolbar or Edit | Paste in the menu bar of the copy of HP TestExec SL that has the destination testplan loaded.

When moving tests or test groups across testplans, the following rules apply to testplan variants:

- If the source and destination testplans have a variant with the same name, tests are simply moved and the variant’s attributes—i.e., its name, parameters, and test limits—are preserved.

- If the source testplan has a variant that does not exist in the destination testplan, the attributes for the variant associated with the test or test group being moved are lost. In other words, a new variant is not created in the destination testplan.

- If the destination testplan has a variant that does not exist in the source testplan, a new variant is created for the test or test group being moved. This new variant assumes the attributes of the first variant in the source testplan, which typically is the variant named “Normal.”

To Copy a Test/Test Group Within a Testplan

1. Select one or more tests or test groups in the left pane of the Testplan Editor window.
2. Choose ⎘ in the toolbar or Edit | Copy in the menu bar.
3. Click the desired new location for the test or test group.

   If you click an existing line, the test or test group will be inserted before that line.
4. Choose 📋 in the toolbar or Edit | Paste in the menu bar.

To Copy a Test/Test Group Across Testplans

1. Run two copies of HP TestExec SL and load the source testplan in one and the destination testplan in the other.
2. Select one or more tests or test groups in the left pane of the Testplan Editor window in the copy of HP TestExec SL that has the source testplan loaded.
3. Choose ⎘ in the toolbar or Edit | Copy in the menu bar of the copy of HP TestExec SL that has the source testplan loaded.
4. Click the desired new location for the test or test group in the left pane of the Testplan Editor window in copy of HP TestExec SL that has the destination testplan loaded.

If you click an existing line, the test or test group will be inserted before that line.

5. Choose 📋 in the toolbar or Edit | Paste in the menu bar of the copy of HP TestExec SL that has the destination testplan loaded.

When copying tests or test groups across testplans, the following rules apply to testplan variants:

- If the source and destination testplans have a variant with the same name, tests are simply copied and the variant’s attributes—i.e., its name, parameters, and test limits—are preserved.
- If the source testplan has a variant that does not exist in the destination testplan, the attributes for the variant associated with the test or test group being copied are lost. In other words, a new variant is not created in the destination testplan.
- If the destination testplan has a variant that does not exist in the source testplan, a new variant is created for the test or test group being copied. This new variant assumes the attributes of the first variant in the source testplan, which typically is the variant named “Normal.”

To Delete a Test/Test Group

1. Select a test or test group in the left pane of the Testplan Editor window.
2. Choose Edit | Delete in the menu bar.

Controlling the Flow of Testing

Using Flow Control Statements

Note

You specify flow control statements in predefined, “fill in the blanks” dialog boxes. If you make an error in entering the syntax, you will be prompted to correct it.

Which Flow Control Statements are Available?

HP TestExec SL supports the following statements that let you control the flow of testing in a testplan. (See the following topic for the syntax of expressions in flow control statements.)

if...then...else

Conditionally executes one or more statements in the testplan, depending upon the value of an expression.

```
if Expression then
  [statements]
[else
  [statements]]
end if
```

Example:

```
if System.RunCount = 5 then
  test Test1
else
  test Test2
end if
```

for...next

Repeats one or more statements in the testplan a specified number of times. A negative value for `Step` causes the counter to decrement.

```
for Variable = Start to End step Step
  [statements]
next
```

Example:

```
for Counter = 1 to 5 step 1
  test Test1
next
```

for...in

Repeats one or more statements in the testplan for each value in a list of arguments.

```
for Variable in Group
  [statements]
next
```

Example:

```
A = 4
B = 2
C = 9
for SequenceLocals.MyVariable in C,A,B
  ! Assume that SequenceLocals.MyVariable
  ! is passed as a parameter to Test1
  test Test1
next
```

loop

Repeats one or more statements in the testplan until a condition specified in an expression is satisfied.

```
loop
   [statements]
   exit if Expression
end loop
```

Example:

```
loop
  test Test1
  test Test2
  exit if SequenceLocals.MyVariable = 3
end loop
```

It also supports the miscellaneous syntax elements listed below, which you can use with the flow control statements.

= (assignment operator)

Sets a variable to a value.

```
Variable = Value
```

Example:

```
X = 2
SequenceLocals.MyVariable = 7
```

comment

Non-executing line used to document a testplan.

Example:

```
! This is a comment
```

else, **end if**, **next**, **end loop**, **exit if**

Syntax elements used with the flow control statements. Some of these are required and others optionally extend the functionality of the flow control statements.

What is the Syntax for Expressions?

Expressions in HP TestExec SL’s flow control statements are a combination of operators, variables, and parentheses that use the BASIC-like syntax described in the following topics.

Using Arithmetic Operators

HP TestExec SL lets you use the math operators listed below in expressions in flow control statements.

Addition The addition operator adds two numbers or concatenates two strings. The operands can be any two numeric expression or strings. However, they must both be of the same type (either numeric or string). If the variables are strings, they will be concatenated.

result = *expr1* + *expr2*

Subtraction The subtraction operator subtracts two numbers. The operands can be any numeric expression.

result = *expr1* - *expr2*

Multiplication The multiplication operator multiplies two numbers. The operands can be any numeric expression.

result = *expr1* * *expr2*

Division The division operator divides two numbers. The operands can be any numeric expressions.

result = *expr1* / *expr2*

Using Relational Operators

HP TestExec SL's relational operators let you compare two string or numeric expressions in flow control statements. They follow the general syntax of

result = *expr1* <**operator**> *expr2* where *result* is any numeric variable. You can use the relational operators listed below to determine if *result* is True (any non-zero value) or False (zero).

Operator	Meaning	True if...	False if...
<	Less than	*expr1* < *expr2*	*expr1* >= *expr2*
<=	Less than or equal to	*expr1* <= *expr2*	*expr1* > *expr2*
>	Greater than	*expr1* > *expr2*	*expr1* <= *expr2*
>=	Greater than or equal to	*expr1* >= *expr2*	*expr1* < *expr2*
=	Is equal to	*expr1* = *expr2*	*expr1* <> *expr2*
<>	Not equal to	*expr1* <> *expr2*	*expr1* = *expr2*

Using Parentheses

If desired, you can use parentheses to force the order of execution of operators in expressions in flow control statements. Parentheses cause individual statements inside the parentheses to be evaluated first. This evaluation is then used with other operators in an expression to evaluate the overall statement.

Refer to the example below, which shows an assignment statement that includes arithmetic operators.

```
A = 2 + 3 * 4
```

Here, the value of A evaluates to 14 because the normal order of operator precedence is multiplication before addition.

The next example uses parentheses in the assignment statement to override the normal order of operator precedence. Here, the value of A evaluates to

20 because the parentheses force the addition of 2 and 3 before multiplying their sum by 4.

```
A = (2 + 3) * 4
```

What Are the Rules for Using Flow Control Statements?

Keep the following in mind when using flow control statements:

- Variable names can be either the name of the symbol by itself, such as “A” or “MySymbol”, or include the name of an internal or external symbol table, such as “SequenceLocals.MySymbol”.

  Note: Variables in flow control statements must use symbols in global symbol tables (SequenceLocals, System, or external).

- If you use a variable in a flow control statement but do not specify a symbol table as part of the variable’s declaration, HP TestExec SL looks for an existing symbol with the same name in the SequenceLocals symbol table. If there is no existing symbol, one is automatically created in SequenceLocals.

Note

Symbols that are created automatically like this default to a numeric data type. If you need a different data type, such as a string, you must explicitly create the symbol prior to using it.

To Insert a Flow Control Statement into a Testplan

1. In the left pane of the Testplan Editor window, choose the desired insertion point in your testplan.

   You can insert a statement on a blank line or into existing tests or statements. If you click to highlight an existing test or statement, the new statement will be inserted immediately preceding it.

2. Choose Insert | Other Statements in the menu bar and select the desired kind of flow control statement.

3. Use the right pane of the Testplan Editor window to enter any declarations required for the specific kind of flow control statement you chose.

Interacting with Flow Control Statements

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.

If desired, you can directly manipulate the value of a variable in a flow control statement or use the variable’s value to control some aspect of testing. Then, examining or modifying the value of the symbol is the same as examining or modifying the value of the variable in the testplan.

How is this useful? Suppose you were testing a module whose stimulus—an input voltage, perhaps—needed to vary within predefined limits until the module either passed or failed. You could:

1. Execute the test for that module in a “for...next” loop, such as:

   ```
   for Voltage = 9.9 to 10.1 step 0.1
     ModuleTest
   next
   ```

2. In the test for the module, query the value of the counter variable and use it to vary the stimulus.

   ```
   ModuleTest
     ...Get value of Voltage from symbol table
     ...Use value of Voltage to increment input voltage
   ```

Other examples of using flow control statements with symbols include:

- Branching on passing or failing tests, which are described under “To Branch on a Passing Test” and “To Branch on a Failing Test”

- Executing a test or test group only once per run of the testplan, which is described under “To Execute a Test/Test Group Once Per Testplan Run”

To Branch on a Passing Test

You can use an “if...then” statement to examine the predefined TestStatus symbol in the System symbol table and programmatically implement an “on pass branch to” feature based on the results of a test; e.g.,

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

1. In the left pane of the Testplan Editor window, click to select the line that follows the test upon which you wish to branch.

   Tip: You can click the line that follows the test even if it is blank.

2. Choose Insert | Other Statements | If...Then in the menu bar.

3. With the “if...then” statement selected in the left pane of the Testplan Editor window, specify “System.TestStatus = 0” for the value of Expression in the right pane.

4. Place any tests, test groups, or statements you wish to have executed as “branch on pass” within the boundaries of the “if...then” statement.

To Branch on a Failing Test

You can use an “if...then” statement to examine the predefined TestStatus symbol in the System symbol table and programmatically implement an “on fail branch to” feature based on the results of a test; e.g.,

```
test Test1
if System.TestStatus >= 1 then
  ! If Test1 failed run Test2
  test Test2
end if
test Test3
```

Or, you can use the graphical On Fail Branch To feature that is built into each test.

Do either of the following:

1. In the left pane of the Testplan Editor window, click to select the line that follows the test upon which you wish to branch.

   Tip: You can click the line that follows the test even if it is blank.

2. Choose Insert | Other Statements | If...Then in the menu bar.

3. With the “if...then” statement selected in the left pane of the Testplan Editor window, specify “System.TestStatus >= 1” for the value of Expression in the right pane.

4. Place any tests, test groups, or statements you wish to have executed as “branch on fail” within the boundaries of the “if...then” statement.

   - *or* -

1. In either the Main or Exception sequence, click a test in the left pane of the Testplan Editor window.

2. Choose the Options tab in the right pane of the Testplan Editor window.

3. Click the arrow to the right of “On Fail Branch To” to invoke a list of tests to which the current test can branch if a failure occurs.

   The default value of “<Continue>” means that if the current test fails, the next test in the list will be executed; i.e., there is no branching.

4. Click a test in the list to select it as the desired branch.

To Branch on an Exception

1. In the left pane of the Testplan Editor window, click the arrow to the right of “Testplan Sequence”.

2. Choose “Exception” in the list.

3. Add one or more tests to the list of tests for the Exception sequence.

This list of tests will be executed if an exception occurs when executing the testplan.

4. Click the arrow to the right of “Testplan Sequence”.

5. Choose “Main” in the list to return to the Main—i.e., non-exception—sequence of tests.

To Execute a Test/Test Group Once Per Testplan Run

You can use an “if...then” statement to examine the predefined RunCount symbol in the System symbol table and have specific tests, test groups, or statements executed only once each time the testplan runs; e.g.,

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

1. In the left pane of the Testplan Editor window, click to select a line where you wish to insert an “if...then” statement to bound one or more tests, test groups, or statements to be executed only once per testplan run.

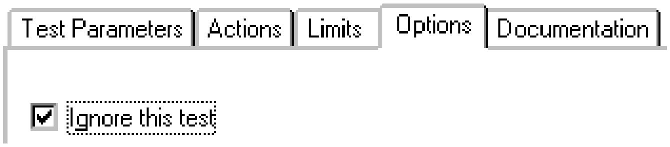
2. Choose Insert | Other Statements | If...Then in the menu bar.

3. With the “if...then” statement selected in the left pane of the Testplan Editor window, specify “System.RunCount = 1” for the value of Expression in the right pane.

4. Place the desired tests, test groups, or statements within the boundaries of the “if...then” statement.

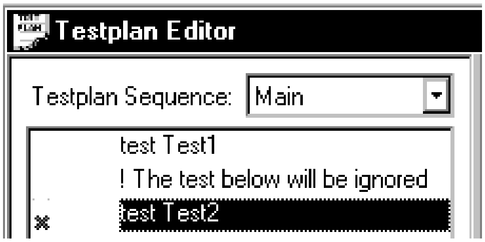
To Ignore a Test

If desired, you can use the “Ignore this test” feature to ignore a test when the testplan is run. Because no integrity checking is done on ignored tests, they are useful when you wish to insert non-working tests during testplan development and finish them later. Also, you can use ignored tests in conjunction with variants so that one variant of a testplan executes different tests than does another variant.



☑ Ignore this test

As shown below, an ignored test has a small cross beside it in the sequence of tests.



Testplan Sequence: Main

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

1. With a testplan loaded, in the left pane of the Testplan Editor window click to select the test to be ignored.

Note

If you are using variants, specify which variant to use before telling the Test Executive to ignore a test. For more information about variants, see "Testplan Variants" in Chapter 3 of the *Using HP TestExec SL* book.

2. Choose the Options tab in the right pane of the Testplan Editor window.

3. Check the box labeled "Ignore this test".

Running a Testplan

To Load a Testplan

Load a testplan so you can examine, modify, or run it.

1. Click 📂 in the toolbar or choose File | Open in the menu bar.
2. Type the name of an existing testplan file (.tpa) or use the graphical browser to find an existing testplan.
3. Choose the Open button.

To Run an Entire Testplan

Run a testplan to execute the tests in it.

1. Load the testplan, if needed.
2. (*optional*) If you wish to use a testplan variant other than the default, Normal, do the following:
   a. Click ℹ in the toolbar or choose Options | Testplan Options in the menu bar.

   b. On the Execution tab in the right pane of the Testplan Editor window, choose the desired variant from the list under Testplan Variant.

      Tip: The current variant is shown toward the right side of the status bar at the bottom of the Test Executive environment.

   c. Choose the OK button.
3. Choose ▶ in the toolbar or choose Debug | Go in the menu bar.

To Run Selected Tests in a Testplan

Run selected tests in a testplan to exercise the functionality of a subset of the testplan.

Caution

Running selected tests is a powerful feature for debugging testplans. However, its degree of safety and effectiveness depends on the setup and cleanup tasks associated with tests. For best results, be sure you either know what state hardware is in before and after each test or use a setup/cleanup action to specify explicitly the entry and exit conditions for each test. If the tests you select to run appear inside a test group, the setup/cleanup tasks associated with the test group will be run.

1. Load the testplan if it is not already loaded.

2. (*optional*) If you wish to use a testplan variant other than the default, Normal, do the following:

    a. Click ⓘ in the toolbar or choose Options | Testplan Options in the menu bar.

    b. On the Execution tab in the right pane of the Testplan Editor window, choose the desired variant from the list under Testplan Variant.

       Tip: The current variant is shown toward the right side of the status bar at the bottom of the Test Executive environment.

    c. Choose the OK button.

3. Select one or more tests by holding down the Ctrl key and clicking them in the left pane of the Testplan Editor window.

   Tip: You can select an inclusive range of tests by holding the Ctrl key and clicking a test at the start of the desired range, holding the Shift key without releasing the Ctrl key, and then clicking a test at the end of the desired range.

4. Do either of the following to run the selected tests:

   - Choose Debug | Run Selected Tests

     - or -

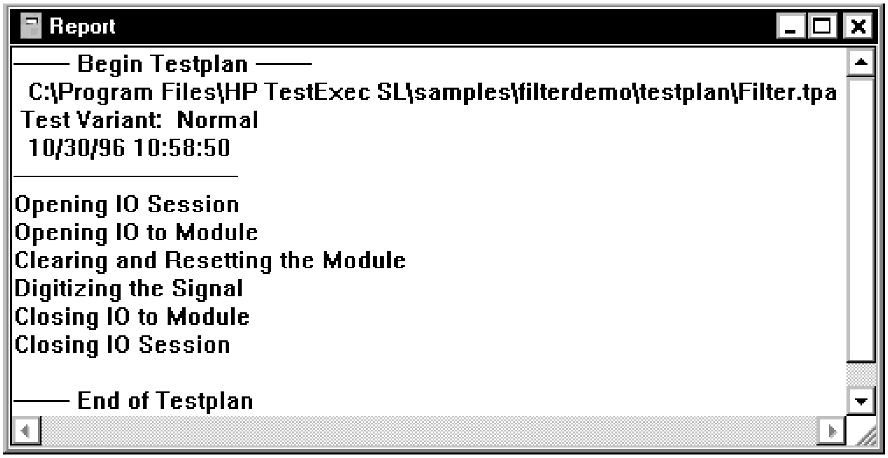
   - Right-click on one of the selected tests and choose “Run Selected Tests” from the menu that appears.

The selected tests will be run in the order in which they appear in the left pane of the Testplan Editor window.

Viewing What Happens as a Testplan Runs

Using the Report Window to Monitor Results

As shown below, the Report window lets you monitor the results as a testplan runs.



```
——— Begin Testplan ———
 C:\Program Files\HP TestExec SL\samples\filterdemo\testplan\Filter.tpa
Test Variant:  Normal
 10/30/96 10:58:50
————————————————
Opening IO Session
Opening IO to Module
Clearing and Resetting the Module
Digitizing the Signal
Closing IO to Module
Closing IO Session

——— End of Testplan
```

Tip: You may want to minimize the Report window if you wish to examine a report later but do not want the Report window appearing all the time.

To Enable/Disable the Report Window

- With a testplan loaded, click 🗎 in the toolbar or choose Window | Report in the menu bar.

A check mark appears to the left of Report in the upper region of the Window menu when the Report window is enabled.

To Specify What Appears in the Report Window

1. With a testplan loaded, click ℹ in the toolbar or choose View | Testplan Options in the menu bar.

2. When the Options box appear, choose its Reporting tab.

3. Enable/disable any or all of the following check boxes under Report.

   Passed tests If enabled, information about tests that pass appears in the Report window.

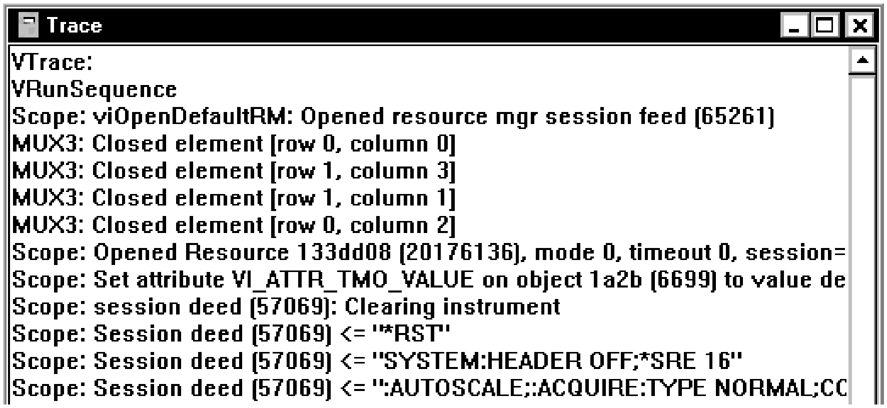
   Failed tests If enabled, information about tests that fail appears in the Report window.

   Exceptions If enabled, information about exceptions that occur while executing the testplan appears in the Report window.

4. Choose the OK button.

Using the Trace Window to Monitor I/O Operations

As shown below, the Trace window lets you dynamically monitor I/O operations with hardware, such as instruments and switching modules, in a test system as a testplan runs. Options associated with it let you specify when to trace tests and how much information to gather during tracing.



```
VTrace:
VRunSequence
Scope: viOpenDefaultRM: Opened resource mgr session feed (65261)
MUX3: Closed element [row 0, column 0]
MUX3: Closed element [row 1, column 3]
MUX3: Closed element [row 1, column 1]
MUX3: Closed element [row 0, column 2]
Scope: Opened Resource 133dd08 (20176136), mode 0, timeout 0, session=
Scope: Set attribute VI_ATTR_TMO_VALUE on object 1a2b (6699) to value de
Scope: session deed (57069): Clearing instrument
Scope: Session deed (57069) <= "*RST"
Scope: Session deed (57069) <= "SYSTEM:HEADER OFF;*SRE 16"
Scope: Session deed (57069) <= ":AUTOSCALE;:ACQUIRE:TYPE NORMAL;CC
```

Trace information appears in named “streams” of information that identify the information’s source. The name of the stream is followed by a semicolon and the status message for that stream. In the example above, MUX3 is the name of a trace stream whose source is a hardware handler that controls a switching module whose logical name is “MUX3”. Status information from MUX3, such as “Closed element [row 0, column 0]”, describes what is happening at MUX3 as the testplan runs. “Scope” is another stream in the example.

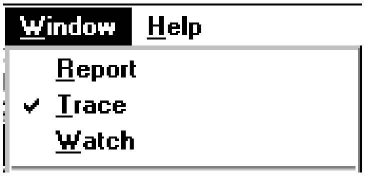
Using the Trace window is a three-step process. You must:

1. Enable the Trace window
2. Specify which tests to trace
3. Specify what kind of trace information to display for each traced test

To Enable/Disable the Trace Window

- With a testplan loaded, choose Window | Trace in the menu bar.

As shown below, a check mark appears to the left of Trace in the upper region of the Window menu when the Trace window is enabled.

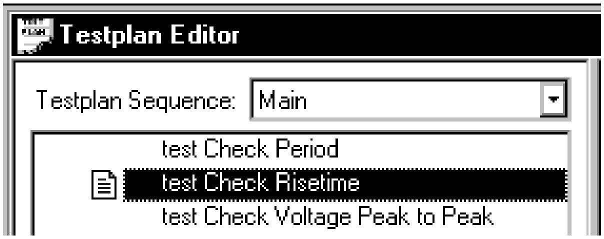


To Specify Which Tests are Traced

1. With a testplan loaded, in the left pane of the Testplan Editor window choose one or more tests to be traced.

2. Choose Debug | Set Trace in the menu bar.

As shown below, a trace icon appears to the left of traced tests.



Tip: A quick way to select all tests for tracing is to choose a test in the left pane of the Testplan Editor window, type Ctrl-a or choose Edit | Select All in the menu bar, and then choose Debug | Set Trace in the menu bar.

To Specify What Appears When Tests are Traced

1. With a testplan loaded, choose Debug | Trace Settings in the menu bar.

2. Enable/disable any or all of the following items under Trace Settings. Each corresponds to a named stream of trace information.

User Trace If enabled, user-defined trace information appears for actions in traced tests as the testplan runs. This is the default stream for trace information sent from actions.

"User-defined trace information" means information programmatically sent to the Trace window from actions via API functions such as `UtaTrace()`. See the *Reference* book for more information about APIs used for tracing.

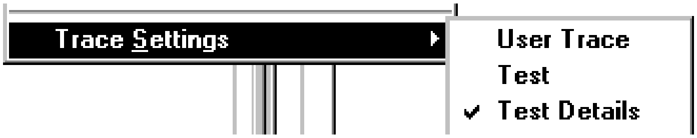
Test If enabled, test names appear for traced tests in the Trace window as the testplan runs.

Test Details If enabled, detailed information about traced tests appears in the Trace window as the testplan runs.

other Some actions, hardware handlers, or instrument drivers add other stream names to the Trace settings menu.

API functions such as `UtaTraceEx()` and `UtaHwModTraceEx()` let you send trace information in named streams from actions and hardware handlers, respectively. See the *Reference* book for more information about APIs used for tracing.

As shown below, a check mark appears next to the names of streams selected for tracing.



To Stop a Testplan

When you stop a testplan, execution halts when the current operation, such as executing an action, has finished.

- Choose Debug | Stop or in the menu bar or ■ in the toolbar.

Note

If you need to halt a testplan immediately, use the Abort command instead.

To Abort a Testplan

When you abort a testplan, execution halts immediately regardless of what the testplan is doing.

- Choose Debug | Abort or in the menu bar or ▣ in the toolbar.

Note

If you wish to complete the current operation in progress—such as executing an action—before halting, use the Stop command instead.

Other Tasks Associated with Testplans

Using Global Variables in Testplans

Global variables let actions share data across tests in a testplan. The scope of a global variable can be:

- The entire testplan, which means the symbol is stored in an external symbol table or in the System symbol table.
- Restricted to a single sequence in a testplan, which means the symbol is stored in the SequenceLocals symbol table.

For detailed information about using symbols tables, see “Using Symbol Tables” in Chapter 5.

Note By default, HP TestExec SL stores some global information in predefined symbols in the System symbol table; see “Predefined Symbols” in Chapter 5.

To Use a Global Variable Whose Scope is the Testplan

1. With a testplan loaded, use the Symbol Tables box (View | Symbol Tables) to declare a new symbol in an external symbol table.

   Example: The name of the external symbol table is “ExtSymTable.sym” and the name of the symbol in it is “ExtSymbol”.

   Note: If there is no existing external symbol table to hold your global variable, use File | New and choose Symbol Table to create a new one. Then choose the Link to External Symbol Table button in the Symbol Tables box to make the externally stored symbol visible to your testplan.

2. Choose the Actions tab in the right pane of the Testplan Editor window.

3. In the list of actions, choose an action that has a parameter you wish to associate with the global variable.

Example: The name of the parameter is “dutvoltage”.

4. Click the Name column in the row that contains the parameter of interest.

5. Click the ... button to invoke the editor for the parameter.

6. When the editor for the parameter appears, enable *references a symbol* if it is not already enabled.

7. Use the Available Tables list to select the external symbol table that contains the global variable you created earlier.

8. Use the Available Symbols list to select the name of the global variable.

9. Choose the OK button.

Example: The value of parameter “dutvoltage” now is “ExtSymTable.ExtSymbol”; i.e., the value of the parameter is determined by the value of symbol “ExtSymbol” in the symbol table named ExtSymTable.

To Use a Global Variable Whose Scope is a Sequence

1. In the left pane of the Testplan Editor window choose a sequence—Main or Exception—in which to use the global variable.

2. If the chosen sequence does not already have a suitable symbol defined in its SequenceLocals symbol table, create one.

   Example: Assume the name of the global variable is “GlobalVar”.

3. Choose the Actions tab in the right pane of the Testplan Editor window.

4. In the list of actions, choose an action that has a parameter you wish to associate with the global variable.

   Example: The name of the parameter is "dutvoltage".

5. Click the Name column in the row that contains the parameter of interest.

6. Click the ... button to invoke the editor for the parameter.

7. When the editor for the parameter appears, enable *references a symbol* if it is not already enabled.

8. Use the Available Tables list to select the SequenceLocals symbol table.

9. Use the Available Symbols list to select the name of the global variable.

10. Choose the OK button.

Example: The value of parameter “dutvoltage” now is “SequenceLocals.GlobalVar”; i.e., the value of the parameter is determined by the value of variable “GlobalVar” in the SequenceLocals symbol table for the chosen sequence.

To Specify the Global Options for a Testplan

1. With a testplan loaded, click ⓘ in the toolbar or choose Options | Testplan Options in the menu bar.

2. Use the features on the various tabs in the Testplan Options box to specify the global options for the current testplan.

To Specify Which Topology Files to Use

1. With a testplan loaded, choose Options | Switching Topology Files in the menu bar.

2. Type the name of a topology file for the fixture layer or click the associated Browse button and use the graphical browser to choose a file.

3. Type the name of a topology file for the UUT layer or click the associated Browse button and use the graphical browser to choose a file.

4. Choose the OK button.

Note Topology files have a ".ust" extension; e.g., "fixture1.ust".

Using Testplans & UUTs with an Operator Interface

To Register a Testplan for an Operator Interface

A typical operator interface lets production operators choose from a list of testplans to run. You must manually edit file "tstexcsl.ini" to specify which testplans appear in the list, which variant is chosen by default, and a brief description of what the testplan does.

1. Open file "tstexcsl.ini" (in directory "<*HP TestExec SL home*>\bin") with a text editor, such as WordPad in its text mode.

2. Add entries for one or more testplans to the [Testplan Reg] section of the file.

Note The file contains descriptive comments about the formats of these entries.

3. Save the updated file and exit the editor.

To Register a UUT for an Operator Interface

Some operator interfaces let production operators use a bar code reader to scan the information for a UUT, and then parse the bar code to automatically load the appropriate testplan. If your operator interface supports this feature, you must manually edit file "tstexcsl.ini" to specify the association between UUTs and testplans.

1. Open file "tstexcsl.ini" (in directory "<*HP TestExec SL home*>\bin") with a text editor, such as WordPad in its text mode.

2. Add entries for one or more UUTs to the [UUT Reg] section of the file.

Note The file contains descriptive comments about the formats of these entries.

3. Save the updated file and exit the editor.

Using Variants in Testplans

Variants let you create named variations on the contents of a testplan. After you create a testplan's variants, you can specify the parameters and limits for the tests and test groups in each variant. 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.

To Add a Variant to a Testplan

1. With a testplan loaded, choose Options | Variants in the menu bar.
2. When the Test Variants box appears, choose the Add button.
3. In the Add Variant box, type a name for the new variant in the field under New Variant.
4. Choose a template for the new variant from the list of existing variants shown under Based On.

   Tip: Base the new variant on whichever existing variant is most like the new one.
5. Choose the OK button.

For information about specifying the contents of variants after you have created them, see "Specifying Variations on Tests/Test Groups When Using Variants" in Chapter 2.

To Rename a Variant in a Testplan

1. With a testplan loaded, choose Options | Variants in the menu bar.
2. When the Test Variants box appears, click the name of an existing variant in the list under Current Variants.
3. Choose the Rename button.

4. In the Rename Variant box, choose the name of an existing variant from the list shown under Variant Name.

5. Type a new name for the variant in the field under New Name.

6. Choose the OK button.

To Delete a Variant from a Testplan

1. With a testplan loaded, choose Options | Variants in the menu bar.

2. When the Test Variants box appears, click the name of an existing variant in the list under Current Variants.

   Note: You cannot delete Normal, which is the default variant.

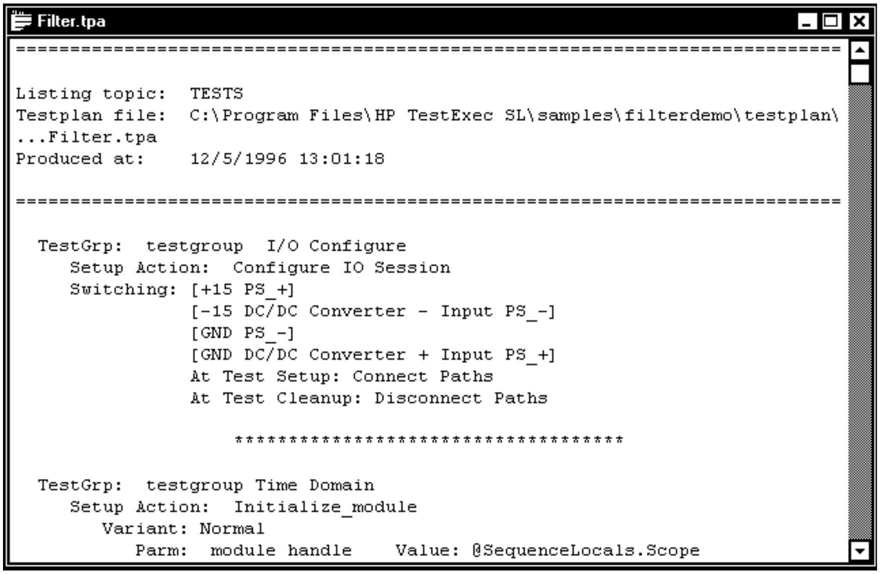
3. Choose the Delete button.

4. Choose the OK button.

Examining Testplans & System Information

Overview

The Listing window lets you view or print information about various aspects of your testplans and hardware controlled by your test system. The example below shows how you can view a descriptive listing of the tests in a testplan.



```
=============================================================================

Listing topic:  TESTS
Testplan file:  C:\Program Files\HP TestExec SL\samples\filterdemo\testplan\
...Filter.tpa
Produced at:    12/5/1996 13:01:18

=============================================================================


  TestGrp:  testgroup  I/O Configure
     Setup Action:  Configure IO Session
     Switching: [+15 PS_+]
                [-15 DC/DC Converter - Input PS_-]
                [GND PS_-]
                [GND DC/DC Converter + Input PS_+]
                At Test Setup: Connect Paths
                At Test Cleanup: Disconnect Paths

                    ************************************


  TestGrp:  testgroup Time Domain
     Setup Action:  Initialize_module
        Variant: Normal
           Parm:  module handle    Value: @SequenceLocals.Scope
```

Which Kinds of Information Can I Examine?

The categories of information you can examine or print in the Listing window include:

Actions	Lists detailed information about actions in the current testplan, including action names, source file names, and routine names
Symbol tables	Lists the symbols used in symbol tables in the current testplan.

Testplan Audit Lists auditing information for the current testplan

Testplan Lists detailed information about the current testplan, including test groups, tests, actions, variants, and run options.

Tests Lists detailed information about tests in the current testplan, including test names, actions, variants, source files names, and routine names.

Adjacencies Lists all topology adjacencies—i.e., nodes separated by a switching element—for the current testplan, including preferred node names, adjacency names, module names, and switching elements and their positions.

Node Labels Lists all node labels for the current testplan, including label names, preferred node names that are aliased, descriptions, and keywords.

Instruments Lists information about instruments controlled by the current testplan.

Switches Lists information about switching hardware controlled by the current testplan.

Fixture Layer Lists topology information about connections on the fixture topology layer, which includes aliases, wires, and modules.

System Layer Lists topology information about connections on the system topology layer, which includes aliases, wires, and modules.

UUT Layer Lists topology information about connections on the UUT topology layer, which includes aliases, wires, and modules.

To List Testplans & System Information

1. Choose View | Listing in the menu bar.

2. Choose which type of listing to view.

To Print Listings of Testplans & System Information

1. Choose View | Listing in the menu bar.

2. Choose which type of listing to view.

3. Click 🖨 in the toolbar or choose File | Print in the menu bar.

4. Set the printing options as desired.

5. Choose the OK button.

Tip: You can use File | Print Preview in the menu bar to see how a listing will look before printing it.

To Find Specific Text in Listings

If desired, you can search any of the various listings of system information for a specific word or phrase.

1. If you have not already done so, generate the listing that you wish to search.

2. With the window in which the listing appears active, click 🔍 in the toolbar or choose Edit | Find in the menu bar.

3. In the "Find what" field, specify the text for which you wish to search.

   Tip: Check the "Match case" box if you wish to search for exactly the same pattern of upper and lowercase characters specified in the "Find what" field.

   Tip: Use the Direction option to choose which direction to search from the current position in the listing.

4. Choose the Find Next button.

To Find Specific Text in Sequences & Lists of Actions

If desired, you can search for a word or phrase in:

- The current sequence that appears in the left pane of the Testplan Editor window

  This lets you find items such as a specific test in a testplan or the name of a specific variable used in a flow control statement.

- The list of action names that appears on the Actions tab in the right pane of the Testplan Editor window

  This lets you find a specific action in a list of actions.

1. With a testplan loaded and the Testplan Editor window active, click in the toolbar or choose Edit | Find in the menu bar to invoke the Find in Sequence/Actions box.

2. In the "Find What" field, specify the text for which you wish to search.

   Tip: Check the "Match Case" box if you wish to search for exactly the same pattern of upper and lowercase characters specified in the "Find What" field.

   Tip: Check the "Whole Word Only" box if you do not wish to find partial forms of the word or phrase.

   Tip: If desired, you can restrict the range of searching by selecting/deselecting options under Search.

3. Choose the Find Next button.

   Tip: The drop-down list under the "Find What" field retains words and phrases used in previous searches. Use this list to quickly respecify a previous search without further typing.

Debugging Testplans

As you develop testplans and their components you need to verify their operation and any fix problems that arise. HP TestExec SL’s debug features let you interact with testplans and their components as they execute.

If you are using C/C++ to develop actions, also see “Debugging C/C++ Actions” in Chapter 3.

Using Interactive Controls & Flags

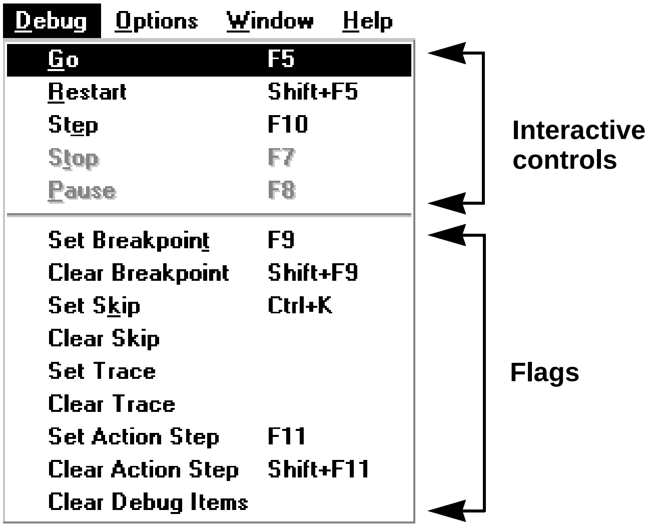
Once started, a testplan normally runs from beginning to end, executing tests in the order in which they appear in it. However, the Test Executive provides several features you can use to modify the running of a testplan. These features can be particularly useful when you are debugging a testplan or test, or when you need to stop or pause the testplan at a specific place while troubleshooting a UUT.

There are two main kinds of features you can use to control testplans:

Interactive Controls These are features such as Stop/Continue, Restart, Step, Stop, and Pause. They are interactive insofar as using them causes an immediate response.

Flags You can set “flags”—i.e., markers—in the testplan. A flag is acted upon if it is encountered as the testplan runs. You can set a flag that marks a test to be stopped upon, skipped, traced, or have its actions single-stepped. Also, you can clear an individual flag or clear all flags for selected tests.

As shown below, these features appear as options under the Debug menu in the menu bar.



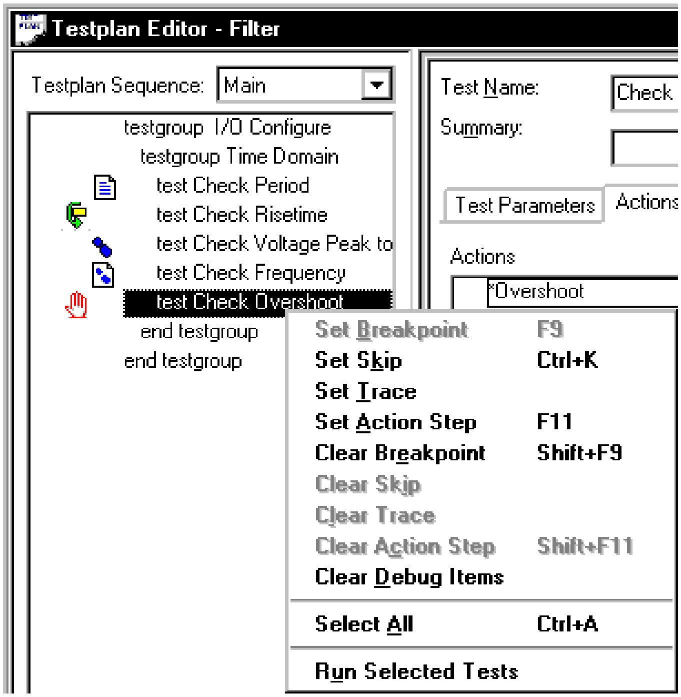
When you use the Debug menu's options to set a flag for a test in a testplan, one of the icons shown below appears to the left of the test.

<u>This icon...</u>	<u>Means that...</u>
✋	A **breakpoint** has been set for the test, which means the testplan will execute until the breakpoint is encountered, and then stop executing immediately before the marked test.
⇩	Items marked in the testplan will be **skip**ped; i.e., the testplan will not execute the marked items. Be aware that skipping a test is not the same as ignoring it (see “Ignoring a Test” earlier in this chapter); the overall integrity of skipped tests is checked, but that of ignored tests is not.
🗎	The test will be **trace**d, which means that status information will appear in the Trace window as the test executes.

Actions in the marked test will be **single-step**ped. The testplan will pause at the first action in the test, and you can use either the Step command in the Debug menu or the icon in the toolbar to execute the test's actions one at a time.

A combination of the **trace** *and* **single-step** icons; i.e., the marked test will be traced as you single-step through it.

As a shortcut when setting flags, you can select a test in the left pane of the Testplan Editor window and then right-click to invoke the menu shown below.



Tip: If desired, you can select multiple tests in a testplan and simultaneously set or clear all of their flags.

Caution

If you add flags and then save a testplan, the flags are saved with it. Be sure to remove flags from testplans before releasing them to production. For example, a breakpoint flag can cause the testplan to stop executing prematurely and leave the operator interface "hung."

Single-Stepping in a Testplan

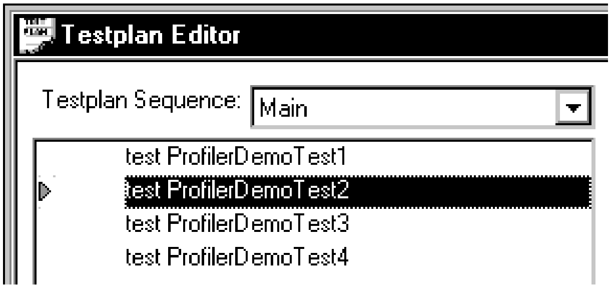
Single-stepping in a testplan lets you pause as needed to verify that tests and actions are working correctly.

Single-Stepping Through Tests

Overview

If desired, you can single-step through the tests in a testplan. Each time you single-step, the testplan executes one test, halts, and then displays a pointer icon that identifies the next test to be executed.

In the example below, test ProfilerDemoTest1 has been executed and the testplan has halted pending execution of test ProfilerDemoTest2.



Testplan Editor

Testplan Sequence: Main

test ProfilerDemoTest1
▷ test ProfilerDemoTest2
test ProfilerDemoTest3
test ProfilerDemoTest4

To Single-Step Through the Tests in a Testplan

- With a testplan loaded, click ▣ in the toolbar or choose Debug | Step Test in the menu bar.

To Cancel Single-Stepping Through the Tests in a Testplan

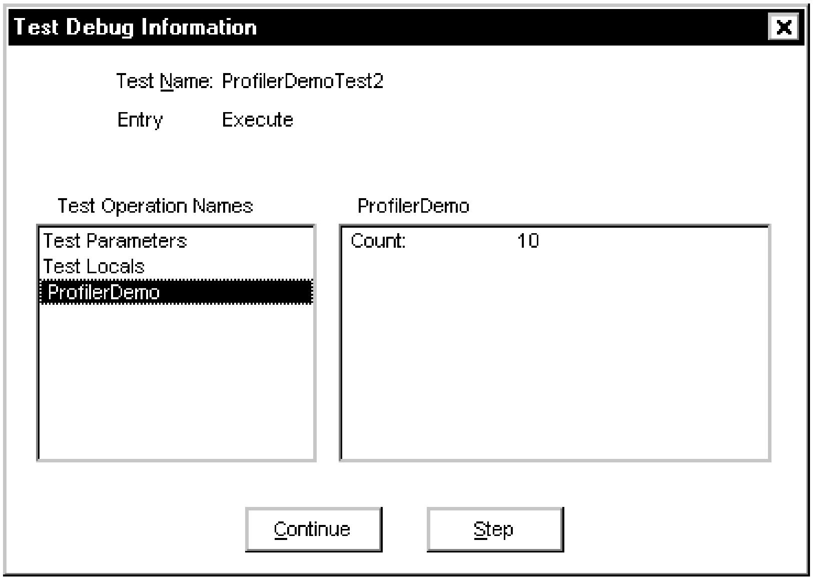
- While single-stepping through a testplan, click ■ in the toolbar or choose Debug | Stop in the menu bar.

Single-Stepping Through Actions

Overview

Each test in a testplan contains one or more actions. If desired, you can single-step through the actions. This can be useful if you wish to verify the results of each action as a test executes. For example, you can connect test equipment to the UUT, pause on a specific action, and verify that the action is interacting correctly with the UUT.

When the testplan is paused while single-stepping through actions, the Test Debug Information box shown below appears.



Test Debug Information

Test Name: ProfilerDemoTest2

Entry Execute

Test Operation Names
Test Parameters
Test Locals
ProfilerDemo

ProfilerDemo
Count: 10

Here, the test’s name is ProfileDemoTest2 and it contains an execute action named ProfilerDemo that uses a parameter named Count whose value is 10. The test is paused on ProfilerDemo.

To Single-Step Through Actions

1. With a testplan loaded, in the left pane of the Testplan Editor window click a test whose actions you wish to step through one at a time.

2. Choose Debug | Set Action Step in the menu bar or right-click and choose Set Action Setup from the menu that appears.

3. Run the testplan as usual.

4. When the test pauses on an action and the Test Debug Information box appears, make debugging measurements or select an item in the list under Test Operation Names and examine its characteristics.

5. Do one of the following:

    - To single-step to the next action in the test (if the test contains more than one action), choose the Step button.

      - *or* -

    - To proceed to the next test without single-stepping through any more actions in the current test, choose the Continue button.

      - *or* -

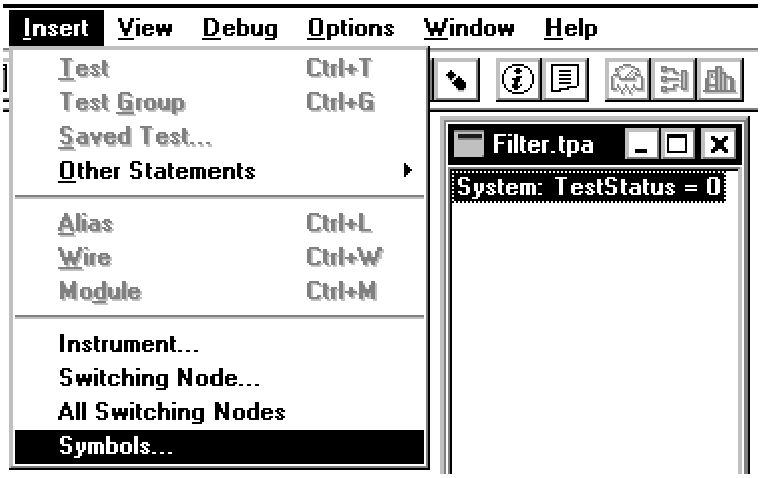
    - To stop after executing the current test, choose ■ in the toolbar and then choose the Continue button.

6. When you have finished single-stepping, clear the flags used to mark the tests.

Using the Watch Window to Aid Debugging

Overview

Many programming environments provide a “watch” feature that lets you examine the values of variables and expressions while debugging programs. In a similar fashion, HP TestExec SL lets you specify items such as symbols, instruments[1], or switching paths to be watched when debugging a testplan. You use the Insert menu to place these items into the Watch window, as shown below, and then examine them when the testplan is paused, such as while single-stepping through actions.



System: TestStatus = 0

The name of the symbol table in which a symbol resides is prefixed to the name of the symbol. In the example above, the symbol named TestStatus appears in the symbol table named System—i.e., System: TestStatus—and its current value is zero.

Note

To ensure that testplans execute rapidly, the Watch window is updated only when testplan execution pauses or stops.

To Insert a Symbol into the Watch Window

1. With a testplan loaded, make sure the Watch window is active; i.e., its border is highlighted.

   Tip: If the Watch window is not visible, choose Window | Watch in the menu bar. If the Watch window is visible but inactive, click its border to make it active.

[1] You can watch instruments only when using specific driver software from Hewlett-Packard.

2. Choose Insert | Symbols in the menu bar.

3. When the Select Symbol to Watch box appears, do the following in it:

    a. Choose a symbol table from the list under Available Tables.

    b. Choose a symbol from the list under Available Symbols.

    c. Choose the OK button.

For more information about symbol tables, see “Using Symbol Tables” in Chapter 5.

To Insert a Switching Node into the Watch Window

1. With a testplan loaded, make sure the Watch window is active; i.e., its border is highlighted.

    Tip: If the Watch window is not visible, choose Window | Watch in the menu bar. If the Watch window is visible but inactive, clicks its border to make it active.

2. Choose Insert | Switching Node in the menu bar.

    Tip: As a shortcut when setting watches on all switching nodes, choose Insert | All Switching Nodes in the menu bar.

3. When the Select Switching Node box appears, do the following in it:

    a. Choose a node from the list.

        Tip: If desired, you reduce the number of nodes that appear in the list by choosing a Filter from the list.

        Tip: If desired, you can sort the list of nodes by selecting the Sort Node Names check box.

    b. Choose the OK button.

For more information about switching nodes, see “About Switching Topology” in Chapter 3 of the *Getting Started* book.

To Insert an Instrument into the Watch Window

Note

This feature is enabled only when using specific instrument drivers provided by Hewlett-Packard.

1. With a testplan loaded, make sure the Watch window is active; i.e., its border is highlighted.

   Tip: If the Watch window is not visible, choose Window | Watch in the menu bar. If the Watch window is visible but inactive, click its border to make it active.

2. Choose Insert | Instrument in the menu bar.

3. When the Select Instrument box appears, do the following in it:

   a. Choose an instrument from the list.

   b. Choose the OK button.

To Remove an Item from the Watch Window

1. In the Watch window, select the item to be removed.

   Tip: If the Watch window is not visible, choose Window | Watch in the menu bar. If the Watch window is visible but inactive, clicks its border to make it active.

2. Choose Edit | Delete in the menu bar.

Fine-Tuning Testplans

A testplan is only as good as the tests in it. Good tests are fast, reliable, and accurate. After you have your tests and testplan running, you may want to consider taking the steps described in the following topics to fine-tune your results.

Optimizing the Reliability of Testplans

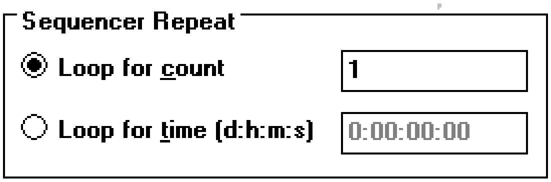
Several ways to improve the reliability of your testplans are:

- Debug known problems in actions and tests as needed.

  For example, you can use the debugging features of the language used to create actions to debug actions. And you can use features in the Test Executive that control the running of testplans to pause on a test, skip a test, and such while debugging tests.

- Run testplans for a prolonged period, such as overnight, to verify the reliability of the tests in them.[1]

  Tip: To run repetitively a testplan, use the “Loop for count” or “Loop for time” options under Sequencer Repeat on the Execution tab in the Testplan Options box (Options | Testplan Options).



Sequencer Repeat

● Loop for count: 1

○ Loop for time (d:h:m:s): 0:00:00:00

- Run testplans with datalogging on and examine the results for consistency.

[1] If you do this, you may want to turn off datalogging to prevent log records from potentially filling your hard disk.

For example, you might turn on datalogging and run the testplan to collect data about a single UUT or a group of UUTs. If the data are inconsistent, try to identify which test(s) is the problem and then fix it.

- Deliberately stress your testplan by introducing conditions that can cause exceptions, and add fixes as needed.

  For example, you might see what happens if an instrument “times out” without returning a reading. Or, you might deliberately test UUTs whose performance is grossly outside the normal limits.

Optimizing the Throughput of Testplans

Suggested Ways to Make Testplans Run Faster

Some ways in which you can make your testplans execute faster are:

- Use test groups to do slow actions outside of tests or to eliminate redundant tasks.

  If you have a group of tests whose setup/cleanup needs are alike, insert those tasks once, at the beginning of a test group that includes the group of tests, instead of inside each test. An example of this might be initializing power supplies or setting up instruments that require similar setups for more than one test. If several tests require positive sources, do the tests as a group. Or, if several tests require the same UUT setting, do the tests as a group.

- Use triggers for fast synchronization of tests.

  For example, avoid synchronizing to slow cycle waveforms. Also, avoid controller-induced test delays.

- Find faster ways to do tests.

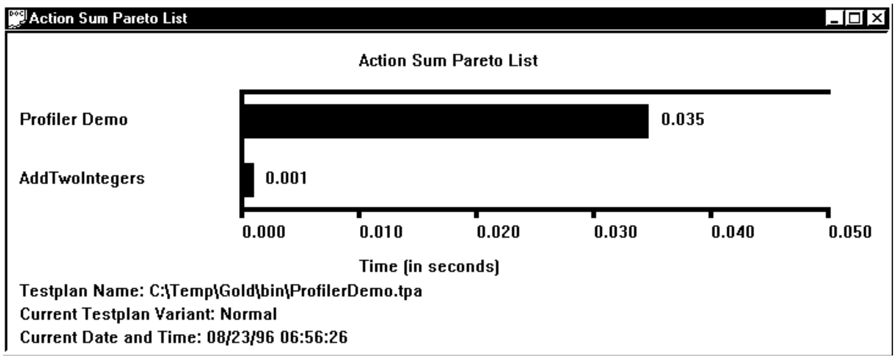
  For example, use a DMM instead of a slower digitizer.

- Use HP TestExec SL’s profiler feature (described below) to optimize the actions inside tests in a testplan.

Using the Profiler to Optimize Testplans

HP TestExec SL includes a profiler 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. As shown below, the profiler display in HP TestExec SL lists actions or test groups in order from slowest to fastest, and shows how long each took to complete.



Testplan Name: C:\Temp\Gold\bin\ProfilerDemo.tpa
Current Testplan Variant: Normal
Current Date and Time: 08/23/96 06:56:26

Each time you run the testplan, profiler data from the previous run is discarded. If a testplan aborts, its profiler data is lost. Also, the profiler is automatically turned off whenever you exit a testplan.

Note Because the profiler can significantly degrade HP TestExec SL’s performance, you probably will not want to run it during production testing.

To Set Up the Profiler

Before you can use the profiler, you must enable it.

1. Choose Options | Testplan Options in the menu bar.

2. In the Testplan Options box, choose the Profiler tab.

3. Enable the Enable Profiler check box.

4. If, besides viewing the profiler data in HP TestExec SL, you want to save the data in a tab-delimited file for subsequent analysis, such as in a spreadsheet, do the following:

   a. Select the Save to File check box.

   b. Either type the name of a file in the data entry field or choose the Browse button and use the graphical browser to specify a name for the file in which the profiling data will be saved.

5. Choose the OK button.

Note

Because a testplan that contains failing tests is likely to have terminated prematurely without executing all of its tests, profiling is automatically disabled when a testplan contains failing tests. If you wish to use profiling anyway, enable the `Ignore all failures` option on the Execution tab of the Testplan Options box (Options | Testplan Options). This causes the testplan to run to completion despite failures.

To Run the Profiler

- With the profiler enabled, run the testplan as usual.

  As the testplan runs with the profiler enabled, HP TestExec SL collects data about the testplan.

To View Profiler Results in HP TestExec SL

1. After running the testplan with the profiler enabled to collect data, choose View | Profiler Results in the menu bar.

2. Choose how you would like to see the data displayed.

Formats for displaying profiler data in Pareto charts include:

Sum of Action Execution Times Total time that actions in the testplan took to execute. If an action is used more than once, this will be its accumulated time.

Average Action Execution Times Average time that actions in the testplan took to execute. If an action is used more than once, this will be the arithmetic mean of each execution time.

Sum of Test Execution Times Total time that tests in the testplan took to execute.

Average of Test Execution Times Average time that tests in the testplan took to execute.

3. If you wish to limit the amount of data that appears, specify an alternate value for Maximum Number of Items to Display.

4. Choose the OK button.

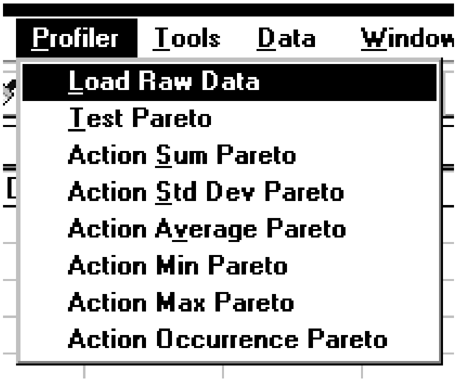
   Tip: If desired, you can simultaneously view other types of Pareto charts by choosing Profiler Pareto from the menu bar and choosing another type when the viewer is active.

   Tip: If desired, you can use File | Print Graph to print the results when the viewer is active.

To View Profiler Results in a Spreadsheet

When you use the profiler’s Write to File option and specify a file name, data is saved in a tab-delimited format suitable for examination with a spreadsheet.

Hewlett-Packard also provides a worksheet (“profile.xls”) and an add-in (“profile.xla”) you can use with Microsoft Excel as the starting point in examining the data file’s contents. These files are located in directory “<*HP TestExec SL home*>\samples\excelmacros”. As shown below, loading either of these files adds a Profiler option and related menu items to Excel's menu bar.



Moving a Testplan

You may want to develop testplans on a central development system that is fully configured even if you intend to use them elsewhere. That way, not every test system needs a full set of hardware resources for compatibility; i.e., each destination system needs only the subset of the development system's resources that are required to run a specific testplan.

Once you have developed and debugged a new testplan on the development system, you probably will want to release it to your production environment. For example, if you intend to run the testplan on more than one test system, you must copy the appropriate files to other systems. Also, you probably will want to make a backup copy of the completed testplan "just in case."

Do the following to move a testplan from your development system to another system:[1]

- Be sure the destination system has all the hardware resources needed to run the testplan.
- Copy the testplan file—i.e., "*testplan_name*.tpa"—to the destination system.
- Be sure all the files used by actions in your testplan exist on the destination system. These include "*.umd" files and executable libraries.

  Tip: You can use View | Listing | Actions to list the contents of actions in a testplan. Or, you can use an audit listing to show all the files used by a testplan.
- Copy the topology files for the fixture and UUT layers ("fixture.ust" and "uut.ust" files or equivalent) to the destination system.
- If external symbol tables are associated with the testplan, copy them ("*.sym" files) to the destination system.

[1] The directory structure on the destination system can be different from the directory structure on the development system.

- Verify that the datalogging options are the same across the systems:
    - Be sure the [Data Log] section in the “tstexcsl.ini” file on the destination system identifies the format and definition files you wish to use when datalogging.
    - Be sure the datalogging options for the testplan (Options | Testplan Options | Reporting) reflect the settings you wish to use on the destination system.
    - Be sure the destination system's topology file for the system layer (“system.ust”) is the same as or a superset of the file on the development system.
    - Be sure to remove any flags, such as skipped tests or breakpoints, if you are moving the testplan to a system used for production testing.

      For more information about flags, see “Using Interactive Controls & Flags.“

Caution Flags left in the testplan can cause the operator interface to behave incorrectly. For example, a breakpoint flag can cause the testplan to stop executing prematurely and leave the operator interface “hung.”

For suggestions about setting up library search paths to optimize the portability of testplans, see “Using Search Paths to Improve Testplan Portability” in Chapter 5.

2

Working With Tests & Test Groups

This chapter describes how to use tests, which are a sequence of actions executed as a group to do some form of testing, and test groups, which are primarily a way of structuring tests.

For an overview of tests and test groups, see Chapter 3 in the *Getting Started* book.

Specifying Parameters for a Test/Test Group

To Add a Parameter to a Test/Test Group

1. With a test or test group selected in the left pane of the Testplan Editor window, choose the Test/Test Group Parameters tab in the right pane.

2. Choose the Insert button.

3. Specify the parameter’s characteristics.

   You can click a row under Value and choose the ... button to invoke a separate editor for the parameter or expand the tree view of the parameter’s characteristics and edit items directly.

   See “Specifying the Properties for Parameters & Symbols” in Chapter 3 of the *Getting Started* book for general information about specifying parameters.

4. Choose the OK button.

Tip: If you enter more than one parameter, you can use the Up and Down buttons to rearrange the order in which parameters appear in the list.

Modifying a Parameter for a Test/Test Group

1. With a test or test group selected in the left pane of the Testplan Editor window, choose the Test/Test Group Parameters tab in the right pane.

2. Choose a parameter in the list under Parameters for Test/Test Group ‘<*name*>’.

3. Choose the Edit button.

4. Modify the parameter’s characteristics.

You can click a row under Value and choose the ... button to invoke a separate editor for the parameter or expand the tree view of the parameter's characteristics and edit items directly.

See "Specifying the Properties for Parameters & Symbols" in Chapter 3 of the *Getting Started* book for general information about specifying parameters.

5. Choose the OK button.

Tip: You can use the Up and Down buttons to rearrange the order in which parameters appear in the list.

To Remove a Parameter from a Test/Test Group

1. With a test or test group selected in the left pane of the Testplan Editor window, choose the Test/Test Group Parameters tab in the right pane.

2. Choose a parameter in the list under Parameters for Test/Test Group '<*name*>'.

3. Choose the Delete button.

4. Choose the OK button.

Specifying Actions for a Test/Test Group

Because actions let tests do useful tasks, tests typically have actions associated with them. Test groups, however, can be useful even without having actions associated with them. For example, you might use test groups simply as aids in structuring your testplans.

To Add an Action to a Test/Test Group

Note Be sure the search paths for action libraries are set up correctly or you may not be able to find the action you want; see “Specifying the Search Path for Libraries” in Chapter 5.

1. With the desired test or test group selected in the left pane of the Testplan Editor window, choose the Actions tab in the right pane.

2. Click in the list under Actions for Test ‘<*test name*>’ to specify where to insert an action into the test.

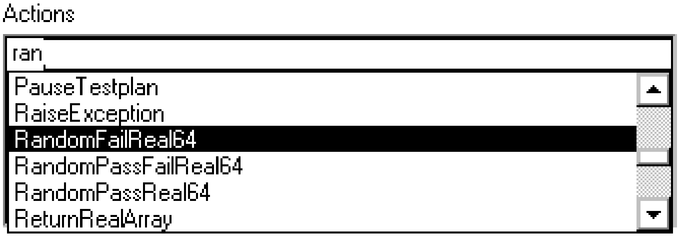
   The action will be inserted immediately before the line selected as the insertion point.

3. Do either of the following:

If the action is a...	Do this...
switching action	a. Choose the Insert Switching button. b. Under Parameters for “Switching”, specify the switching path(s) and what action should be taken for them at setup and cleanup. See “Controlling Switching During a Test” for detailed information about creating switching actions.
regular action	a. Choose the Insert button. b. When the Insert Action box appears, use it to find the desired action and insert it into the test. For more information about using the Insert Action box’s search features, see “Searching for Items in a Library” in Chapter 5. c. Specify the action's parameters and limits (if it returns a result) as needed. See “Specifying the Properties for Parameters & Symbols“ in Chapter 3 of the *Getting Started* book for general information about specifying parameters. Specific procedures for specifying parameters and limits are described in the next couple of topics.

Tip: As a shortcut when inserting regular actions, you can select the area in which the list of actions appear and begin typing. As shown below, the first action whose name matches the characters you type will be selected in the list that appears. Select an action in the list and press Enter to insert it into your test.

Tip: As a shortcut when inserting regular actions, you can select a point in the list of actions and begin typing. As shown below, the first action whose name matches the characters you type will be selected in the list that appears. Select an action in the list and press Enter to insert it into your test.



The list of actions that appears in this box is the same as the list that appears in the Insert Action box. Like the list of actions in the Insert Action box, the contents of this list depend upon the setting for search paths.

Tip: Choose the Move Up and Move Down buttons to rearrange the order in which actions appear in the list.

Tip: Choose the Details button to examine the action’s definition.

To Specify Parameters for Actions in a Test/Test Group

1. With a test or test group selected in the left pane of the Testplan Editor window, choose the Actions tab in the right pane.

2. Click to select an action in the list under Action for Test ‘<*test name*>’.

3. Edit the characteristics of a parameter in the list under Parameters for ‘<*action name*>’.

   You can click a row under Value and choose the ... button to invoke a separate editor for the parameter or expand the tree view of the parameter’s characteristics and edit items directly.

   Tip: If the name of a parameter is italicized, its associated value is the default specified when the action was created. If it is not italicized, the default value has been overridden by a new value.

Tip: If the name of a parameter appears as bold text, that parameter returns a result.

Tip: Right-clicking on a parameter invokes a menu from which you can edit the parameter's value, associate the parameter with a symbol in a symbol table, or reset the parameter's value to its default.

Tip: An @ sign precedes the names of parameters that reference items in symbol tables.

To View Parameters for Actions in a Test/Test Group

1. In the left pane of the Testplan Editor window, select a test or test group.

2. Choose the Actions tab in the right pane of the Testplan Editor window.

3. Click an action in the list under Actions for Test '<*test name*>'.

4. Examine the parameter names and values that appear in the list under Parameters for '<*action name*>'.

   You can click a row under Value and choose the ... button to invoke a separate editor for the parameter or expand the tree view of the parameter's characteristics and examine items directly.

To Specify the Limits for a Test

Note

Although you can use an execute action in a test group, an execute action in a test group cannot return a result for limits checking. Only tests can be used for limits checking.

1. In the left pane of the Testplan Editor window, select the test for which you wish to set limits.

2. Choose the Actions tab in the right pane of the Testplan Editor window.

3. Verify that the execute action chosen to return results for limits checking in the list under Actions for Test '<*test name*>' is the one you want. If it

is not, select the correct one, right-click on it, and choose “Limit check this measurement” from the menu that appears.

Note: An asterisk (*) appears in front of the name of the execute action chosen for limits checking.

4. With the desired action selected on the Actions tab, choose the Limits tab.

5. If you wish to specify a different limits checker for the action that returns results for the test, choose another from the drop-down list to the right of “Limit Checker”.

6. Specify the value(s) for the limits.

   You can click a row under Value and choose the ... button to invoke a separate editor for the limit or edit each limit directly.

   Tip: Right-clicking a limit’s name invokes a menu from which you can edit the limit’s value, associate the limit with a symbol in a symbol table, or reset the limit’s value to its default.

   Tip: As a shortcut when specifying parameters and limits, you can choose the Limits button on the Actions tab and quickly switch to a view of the action’s limits without leaving that tab. To return to viewing the action’s parameters, click the action in the list of actions.

To Remove an Action from a Test/Test Group

1. With a test or test group selected in the left pane of the Testplan Editor window, choose the Actions tab in the right pane.

2. Click to select an action in the list under Action for Test ‘*<test name>*’.

   Tip: You can select an inclusive range of actions by holding the Ctrl key and clicking at the start of the desired range, holding the Shift key without releasing the Ctrl key, and then clicking at the end of the desired range. Alternatively, you can select multiple items by holding the Ctrl key and clicking individual actions.

3. Choose the Delete button.

To Save a Test Definition in a Library

Saving test definitions in a library lets you reuse them as needed, which reduces the amount of work required to create new testplans.

1. With a test selected in the left pane of the Testplan Editor window, choose File | Save Test Definition in the menu bar.

2. In the Test Name field of the Save a Test Definition box, specify a name for the test.

3. If the test has multiple variants, use the list to choose the variant for the version you wish to save.

4. *(optional)* In the Author's Name field, enter the name of whoever created the test.

5. *(optional)* Enter the test's version number, if it has one.

6. *(optional)* Enter a description of the test.

7. *(optional)* Select one or more keywords, one at a time, in the list under Available and choose the Add button to copy them to the list under Selected Keywords.

   Tip: If desired, you can click the blank area in the list under Selected Keywords and create new keywords by typing them there.

   See "How Keywords Simplify Finding Items in Libraries" in Chapter 5 for more information about keywords.

8. Choose the OK button.

9. When the Save As box appears, specify a file name in which to save the test.

10. Choose the Save button.

Note

Although entering optional information is more work initially, it can save time when you reuse code. For example, knowing the author’s name tells you who to contact if you have a question about the test. Or, being able to search by keyword makes it easier to find specific tests later.

To Pass Results Between Tests/Test Groups

If desired, you can pass the results from one test or test group to another test or test group. A result is passed as a parameter to an action.

1. With a testplan loaded, in the left pane of the Testplan Editor window select the test or test group from which you wish to pass results.

2. Choose View | Symbol Tables in the menu bar.

3. When the Symbol Tables box appears, use it to declare a new variable in either the SequenceLocals symbol table or in an external symbol table.

   Note: If you use the SequenceLocals symbol table, be sure the sequence shown in the left pane of the Testplan Editor window is the desired one. You cannot use SequenceLocals to pass results between sequences.

   For information about declaring variables, see “Specifying the Properties for Parameters & Symbols” in Chapter 3 of the *Getting Started* book. For information about the mechanics of using symbol tables, see “Using Symbol Tables” in Chapter 5.

4. In the right pane of the Testplan Editor window, select an action (in the list under Actions for Test ‘<*test name*>’ on the Actions tab) that has a parameter you wish to pass from the test or test group selected in the left pane.

5. Click the Name of the parameter in the list under Parameters for ‘<*action name*>’.

6. Click the ... button to invoke the editor for the parameter.

7. When the editor appears, enable *references a symbol* if it is not already enabled.

8. Use the Available Tables list to select the symbol table that contains the shared variable you created earlier.

9. Use the Available Symbols list to select the name of the shared variable you created earlier.

10. Choose the OK button.

11. In the left pane of the Testplan Editor window, select the test or test group that is to receive the results.

12. In the right pane of the Testplan Editor window, select an action (in the list under Actions for Test ‘<*test name*>’) with a parameter that is to receive the passed value.

13. Click the Name of the parameter in the list under Parameters for ‘<*action name*>’.

14. Click the ... button to invoke the editor for the parameter.

15. When the editor appears, enable *references a symbol* if it is not already enabled.

16. Use the Available Tables list to select the symbol table that contains the shared variable.

17. Use the Available Symbols list to select the name of the shared variable being passed.

18. Choose the OK button.

To Share a Variable Among Actions in a Test/Test Group

Declaring a variable whose scope is a test or test group lets actions inside the test or test group share that variable.

1. With a testplan loaded, in the left pane of the Testplan Editor window select the test or test group whose actions are to share a local variable.

2. Choose the Edit Symbols button on the Actions tab in the right pane of the Testplan Editor window.

3. When the Symbols for Test/Test Group ‘<*test/test group name*>’ box appears, choose its Insert button.

4. Specify the characteristics of the new, shared variable.

   For information about declaring variables, see “Specifying the Properties for Parameters & Symbols” in Chapter 3 of the *Getting Started* book.

5. Choose the OK button in the Symbols for Test/Test Group ‘<*test/test group name*>’ box.

6. Do the following for each action that contains a parameter you wish to have share the newly defined variable:

   a. Select the desired action in the list under Actions for Test/Test Group ‘<*test/test group name*>’ on the Actions tab.

   b. Click the Name of the parameter in the list under Parameters for ‘<*action name*>’.

   c. Click the ... button to invoke the editor for the parameter.

   d. When the editor appears, enable *references a symbol* if it is not already enabled.

   e. Select TestStepLocals from the Available Tables list.

f. Use the Available Symbols list to select the name of the variable you created earlier.

g. Choose the OK button.

Controlling Switching During a Test/Test Group

Switching is dependent upon switching topology, which defines a test system's switchable connections. Switching topology is explained in Chapter 4.

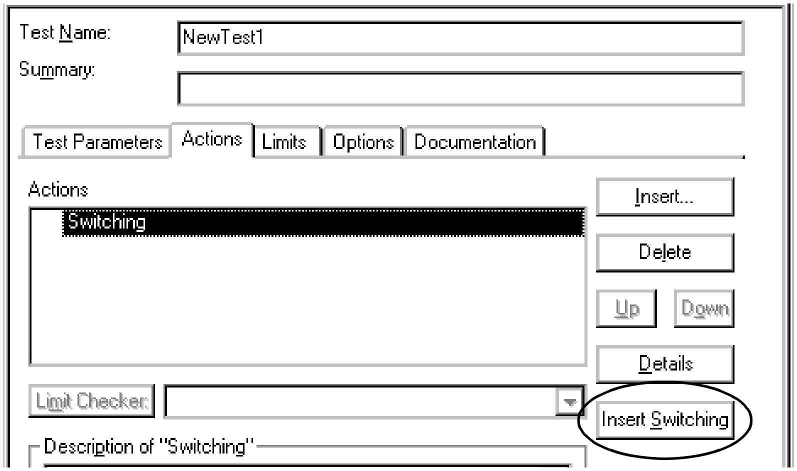
Note

The Throughput Multiplier feature, which lets you use a single set of hardware resources and a single testplan to test multiple UUTs, enhances the functionality of switching actions beyond what is described below. For more information about the effects of the Throughput Multiplier on switching actions, see Chapter 10.

Overview of Creating a Switching Action

If you are using hardware handler software to model your test system's switching hardware and you have used the Switching Topology Editor to describe your topology to the Test Executive, you can:

1. Use the upper section of the right pane of the Testplan Editor window to insert a switching action into your test or test group, as shown below.



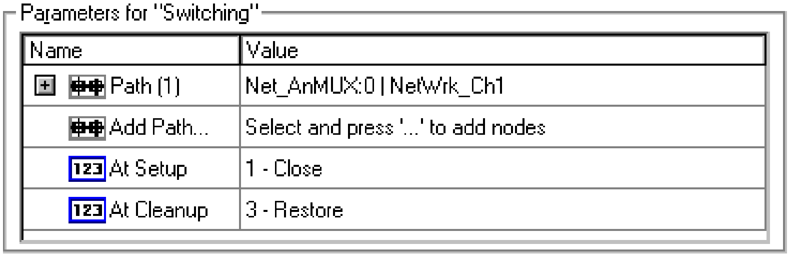
Test Name: NewTest1

Summary:

Actions: Switching

2. Use the lower section of the right pane of the Testplan Editor window to specify the connections needed for the test or test group as well as what

action the switching paths should take at the beginning and end of the test or test group: open, close, or restore the previous state.



Parameters for "Switching"

Name	Value
⊞ Path (1)	Net_AnMUX:0 | NetWrk_Ch1
Add Path...	Select and press '...' to add nodes
At Setup	1 - Close
At Cleanup	3 - Restore

To Create a Switching Action

1. In the left pane of the Testplan Editor window, select the test or test group to which you wish to add a switching action.

2. Choose the Actions tab in the right pane of the Testplan Editor window.

3. Click the desired insertion point in the list under Actions for Test/Test Group ‘<*test/test group name*>’.

   Tip: Use the Up and Down buttons to rearrange items in the list of actions.

4. Choose the Insert Switching button on the Actions tab.

5. Repeat the following steps for each switching path you wish to add to the switching action:

   a. Click the field labeled “Select and press '...' to add nodes” to the right of Add Path under “Parameters for ‘Switching’”.

   b. Click the ... button to invoke the Path Editor.

   c. Use the Path Editor to specify the switching path.

      For more information about specifying switching paths, choose the Help button in the Path Editor

6. Choose the OK button in the Path Editor.

7. Choose an option for At Setup to specify what happens to this group of switching paths when the test or test group begins:

Connect Paths	Switching paths will be closed
Disconnect Paths	Switching paths will be opened

8. Choose an option for At Cleanup to specify what happens to this group of switching paths when the test or test group ends:

Connect Paths	Switching paths be closed
Disconnect Paths	Switching paths will be opened
Undo Setup Paths	Switching paths will be restored to the state they were in prior to the test or test group; i.e., whatever was done for At Setup will be undone

To Delete a Switching Action

1. In the left pane of the Testplan Editor window, select the test or test group that contains a switching action you wish to delete.

2. Choose the Actions tab in the right pane of the Testplan Editor window.

3. Click the switching action to be removed from the list under Actions for Test/Test Group ‘<*test/test group name*>’.

4. Choose the Delete button on the Actions tab.

To Modify a Switching Path in a Switching Action

1. On the Actions tab in the right pane of the Testplan Editor window, select a switching action by clicking it in the list under Actions for Test/Test Group ‘<*test/test group name*>’.

2. Modify the switching path as needed. For any existing path, you can:

   a. Click in the field under Value to the right of the name of the path

   b. Click the ... button to invoke the Path Editor.

   c. Use the Path Editor to modify the path.

      For more information about specifying switching paths, choose the Help button in the Path Editor

To Delete a Switching Path in a Switching Action

You delete an existing switching path by setting its value to nothing.

1. On the Actions tab in the right pane of the Testplan Editor window, select a switching action by clicking it in the list under Actions for Test/Test Group ‘<*test/test group name*>’.

2. Click in the Name field under Parameters for Action ‘<*action name*>’ for the switching path to be deleted.

3. Press the Del key on your keyboard.

Specifying Variations on Tests/Test Groups When Using Variants

Overview

Each variant of a testplan lets you create a potentially unique variation on the parameters and limits associated with the tests and test groups in that variant. 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.

The general procedure for specifying the characteristics of tests or test groups when using variants is:

1. Choose a testplan variant
2. Choose a test or test group
3. Specify the characteristics of the test or test group for that testplan variant

Later, when running the testplan, you can specify which variant to use.

To Specify a Test/Test Group's Characteristics for Each Variant

Repeat the following steps for however many variants your testplan has:

1. Choose Options | Testplan Options in the menu bar.
2. On the Execution tab in the Testplan Options box, choose a variant from the list under Testplan Variant and choose the OK button.
3. Select a test or test group in the left pane of the Testplan Editor window.
4. Use the features on the tabs in the right pane of the Testplan Editor window to specify the characteristics for this variation of the test or test group.

Tip: If you would simply like to examine the characteristics of tests or test groups for each testplan variant, follow the steps above but do not change anything.

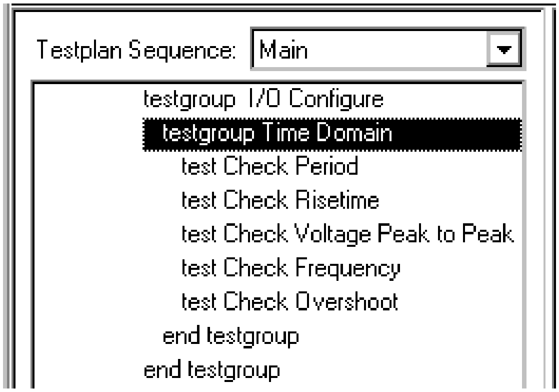
For a conceptual overview of variants, see “Testplan Variants” in Chapter 3 of the *Getting Started* book. For information about creating or modifying variants of testplans, see “Using Variants in Testplans” in Chapter 1.

Viewing the Test Execution Details

Overview

The Test Execution Details window lets you view the details of what will happen when a test or test group is executed. The routines inside actions are listed in the order in which they are executed.

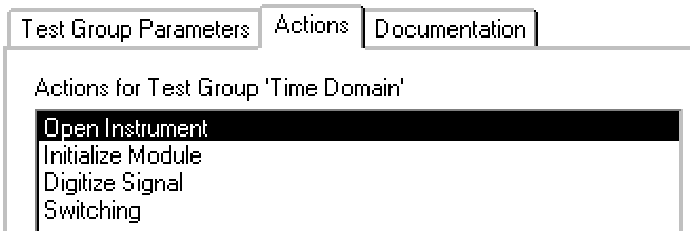
Refer to the example of a testplan shown below.



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
```

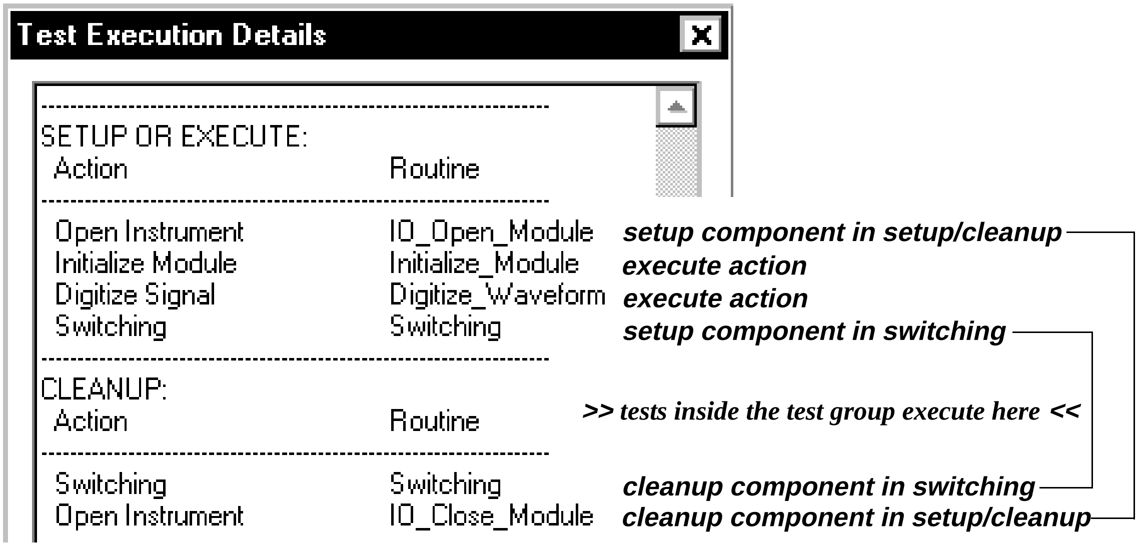
Suppose the following list of actions appeared in the test group named “Time Domain.”



Actions for Test Group 'Time Domain'

Open Instrument
Initialize Module
Digitize Signal
Switching

An annotated example of how the Test Execution Details window would look when examining this test group looks like this:



Test Execution Details

```
-----------------------------------------------------------------
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
```

setup component in setup/cleanup

execute action

execute action

setup component in switching

>> tests inside the test group execute here <<

cleanup component in switching

cleanup component in setup/cleanup

The window contains two columns. The left column lists the names of actions in the test or test group, and the right column lists the names of routines in those actions. The information is further organized into rows that list the actions and their components in the order in which they are executed.

The Test Execution Details window shows that the test group in the example contains four actions: `Open Instrument`, `Initialize Module`, `Digitize Signal`, and one switching action. The action named `Open Instrument` is a setup/cleanup action because it is listed under both SETUP OR EXECUTE and CLEANUP. Both `Initialize Module` and `Digitize Signal` are execute actions because they appear only under SETUP OR EXECUTE, and not under CLEANUP.

Notice how it is implied that the test group's setup components and execute actions are executed before the tests inside the test group, and that the test group's cleanup components are executed afterward. Although the execution details of tests inside the test group do not appear in this listing, you could select those tests individually and view their test execution details.

To View the Test Execution Details

1. With a testplan loaded, click a test or test group in the left pane of the Testplan Editor window.

2. Choose View | Test Execution Details in the menu bar.

3. When you have finished examining the details of the test or test group, choose the OK button.

3

Working With Actions

This chapter describes how to use actions, which are components used to create tests.

For an overview of actions, see Chapter 3 in the *Getting Started* book.

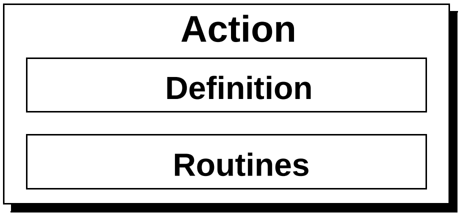
Things to Know Before Creating Actions

Note The topics in this section apply to all types of actions. Subsequent sections describe how to create actions in specific programming languages.

Note HP TestExec SL includes some predefined actions for your use. Look in directory “<*HP TestExec SL home*>\Actions” for subdirectories whose names indicate the kinds of actions they contain, such as math or string manipulation functions. To learn more about these actions, look under Predefined Actions” in the table of contents for HP TestExec SL’s online help.

How Do I Create Actions?

An action consists of two discrete components: a definition that describes the action’s characteristics to the Test Executive environment[1], and action routines (code) that each do one or more tasks.



Given the model above, creating an action is a two-part process:

1. Creating the action definition.

[1] It may help if you think of defining an action as using the Action Definition Editor to “register” the action with the Test Executive.

You use the Action Definition Editor to define the action's characteristics and identify (but not write) the underlying code associated with the action. Each action definition contains the following information:

- The action "style," which adjusts the Action Definition Editor's behavior to match your choice of programming language.
- The name of the action.
- The name of the DLL or other library file in which the action's executable code resides.
- The name of the action's author.
- A description of the action.
- Keywords that help when searching for the action if someone wishes to reuse it later.
- The type of action routine—execute or setup/cleanup. (Typically, most of the actions you use will be execute actions.)
- Definitions of parameters used in the action, including their data types, default values, and descriptions.

2. Creating the action routines.

   You use the editor, debugging tools, and environment of your chosen programming language to write the code for action routines.

In most cases, you can do these two main steps in any order. For example, you may prefer to write the action routine first and then create a definition for it later.

Which Languages Can I Use to Create Actions?

You can write action code in:

- Visual C++ Version 2.0 or higher, 32-bit versions only. (Highly recommended)

- Borland C++, Version 4.0 or higher, 32-bit versions only.
- HP VEE, Version 3.2 or higher for Windows 95.
- National Instruments LabVIEW, Version 4.0 or higher for Windows 95.
- HP BASIC for Windows 6.3.x.

Note You can freely mix actions in various languages so long as they do not access the same instruments within the same testplan. This restriction is necessary because each language is unaware of the other. For example, suppose an action written in C sets an instrument to a particular state. Because it operates in a separate environment, a subsequent action written in HP VEE would be unaware of that state and might inadvertently change it. And, of course, if another C action followed the HP VEE action, it would not be aware of any changes made in HP VEE.

Improving the Reusability of Actions

Designing for Reusability

HP TestExec SL has features that help you reuse action definitions and action routines. To maximize the potential for reusing actions, keep the following in mind when creating them:

- Use a directory structure to organize similar actions into libraries.

  For more information about libraries, see “Using Test & Action Libraries” in Chapter 5.

- Use the Action Definition Editor’s documentation features—keywords, action naming, action descriptions, and parameter descriptions—as an aid to making actions easy to find and use in each library.

- Reuse or modify an existing action whenever possible. Write new actions *only* when no other existing action will work.

Note

You can add new parameters to an existing action and have existing tests that use that action continue to work. Simply specify a default value for each new parameter. Because existing tests will not override the default values of new parameters, the modified action will mimic its previous behavior.

- Short actions that do a single task have greater reusability than more complex actions. When possible, break larger test operations into a shorter series of simple actions.

- Design commonly used actions for use by multiple test sequences. For example, if you have more than one test sequence that requires setting up a digital-analog converter, you could create a separate action that does the converter setup. You could then use that setup action in each of the test sequences that use the converter.

- Use hardware handler software whenever possible (described in Chapter 4).

Documenting Your Actions

Choosing Names for Actions

Each action consists of a definition file and a file that contains the action's executable code.[1] Having a sensible and consistent naming convention helps you organize and describe actions, which makes them self-documenting to some extent. For example, you might use the convention of combining the action name with the step where the routine will be used in the action, such as “MyAction_Execute” or “MyActionSetup”. Or you could use the name of the action to describe what the action does, such as “TrigVolt” for an action that triggers a voltage source or “MeasVolt” for an action that measures a voltage.

For consistency, we recommend that you give the definition file the same name as the action, followed by the extension “.umd”—for example, “DMMSetup.umd”. Then name the code file in accordance with the action’s function, followed by whichever extension is appropriate for the language in

[1] The code can reside in a library that also contains code for other actions.

which the action is written.[1] An example of this might be “DMMSetup.dll” for an action written in C.

Entering Descriptions for Actions

The Action Definition Editor lets you enter a textual description of each action. The description should contain such information as:

- A description of what the action does.
- The action's context, such as whether it is doing a setup, execute, or cleanup function.
- A list of any limitations.
- A list of any special instructions, such as required switching or accompanying actions.

For example, if you had an action that named “adcConfArm”, you could add the description, “Configures the arming subsystem of the analog to digital converter.”

Entering Descriptions for Parameters

You can use the Action Definition Editor to add a textual description to each parameter in the definition of an action. In the description, you should tell what the parameter does, its units of measure, and its range of valid values.

Choosing Keywords for Actions

As you create actions, you will probably store them in libraries from which they will be used to create tests in the Test Executive environment. By letting you associate one or more searchable keywords with each action, the Action Definition Editor helps you quickly locate actions in libraries.

[1] If you create actions in HP BASIC for Windows, all the actions for a given testplan must reside in a single file (server program). You may wish to give that file the same name as the testplan with which it is used.

The keyword feature works best when you follow these rules:

- Always assign keywords to actions. This speeds up the search features in the Test Executive environment.
- Use keywords from the predefined master list of keywords whenever possible. Adding too many keywords increases the length of the search list, which makes it harder to find a specific action. In general, you should have fewer keywords than actions.
- Add a keyword to the master list only if you can use it for other actions.
- If you must create a new keyword, make sure the keyword is meaningful and that it describes the action.

To Define an Action

Use the Action Definition Editor to create an action definition. The general procedure for defining an action in all supported programming languages is described below. Subsequent topics describe the nuances of defining actions in specific languages.

1. Choose File | New in the menu bar.

2. Choose “Action Definition” from the list.

3. Choose the OK button.

4. Choose an action style from the list, which contains:

DLL Style	Action is written in in C/C++
HP VEE	Action is written in HP VEE
LabVIEW	Action is written in National Instruments LabVIEW
HP RMB	Action is written in HP BASIC for Windows

5. Choose the OK button.

6. In the Name field, type the name of the action.

   Tip: Choose a meaningful name that will help when you search for the action later.

7. *(optional)* In the Author field, type your name to identify you as the action's author.

8. In the Library Name field, type the name of the executable library file (such as a DLL) that contains the action routines associated with this action.

Note Leave the Library Name field blank if you are defining an action created in HP BASIC for Windows.

9. *(optional)* In the Description field, type a description of the action.

   Tip: A useful description tells what the action does, gives the context in which the action is used (such as whether it is doing a setup, execute, or cleanup function), lists any limitations, and includes any special instructions, such as required switching or accompanying actions.

10. *(optional)* Do either of the following to make it easier to locate the action in an action library:

    a. Select a master keyword from the predefined list.

    b. Choose the Add button adjacent to the list of keywords to add the keyword to the action.

    - *or* -

    a. If none of the existing master keywords fits the action, type a new keyword in the keyword field.

    b. Choose the Add button adjacent to the list.

    c. If this keyword will be useful with other actions, choose Edit | Add Master Keyword in the menu bar.

11. Choose the Setup/Cleanup or Execute button to specify which kind of action you are defining.

12. Type the names of one or more routines, functions, or subprograms associated with this action (Setup, Execute, & Cleanup fields).

    Tip: Useful, descriptive function names often combine the action name with the step where the function will be used in the action, such as “MyAction_Execute”.

13. Add parameters as needed by choosing the Add button at the bottom of the Action Definition Editor and specifying their properties.

    For more information about specifying parameters, see “Using Parameters with Actions.”

14. *(optional)* Specify auditing information by choosing File | Revision Information in the menu bar and entering descriptive information in the Action Revision Information box.

15. When you have finished defining the action, choose File | Save in the menu bar, specify a name for the new action definition, and save it.

Note Not all parameter types can be used with all programming languages. Any restrictions are noted in the topics that describe how to create actions in specific languages.

Note If you choose “HP VEE” as the action style, an additional Debug check box appears. Checking this box lets you start HP VEE in debug mode so you can debug actions created using HP VEE. After you have finished debugging your actions, unselect this box to return to HP VEE's run-time mode.

Using Parameters with Actions

The topics in this section describe the data types supported when passing data in parameters to actions and the mechanics of using the Action Definition Editor when working with parameters.

Types of Parameters Used With Actions

Each time an action is executed, the Test Executive can pass it one or more parameters or a pointer to a group—called a “block”—of named parameters. Passing specific parameters or a parameter block to the routines in an action creates a unique instance of the action. For example, an action that sets up a power supply might be passed parameters that define voltage and current settings.

Overall, the Action Definition Editor supports these types of parameters for actions:

Type	Description
Complex	Real — The real or magnitude component of a complex number. Imaginary—The imaginary or vector component of a complex number.
Inst	The identifier for an instrument.
Int32	A 32-bit integer.
Int32Array	An array of 32-bit integers.
Node	(*reserved for future use*)
Path	A Switch Configuration Editor path name representing a single switching path.
Point	A pair of 64-bit real numbers, consisting of an X value and a Y value.
PointArray	An array of point data types, where each element consists of an X value and a Y value.
Range	A means of storing data that has a beginning, an end, and an incremental step size, such as frequency sweep data.
RangeArray	An array of ranges.
Real64	A 64-bit real number.
Real64Array	An array of 64-bit real numbers.
Real64Expr	The value of a 64-bit real expression.
String	A group of characters that make up a string.
StringArray	An array of strings.

For more information about data types and how they are used, see Chapter 1 in the *Reference* book.

Note Which specific parameter types you can use in an action definition depends upon which action style you choose. Action styles are described in greater detail later.

Properties you can define that are associated with parameters include:

Value Sets literal values for parameters.

Reference Selects a value by referencing the name of a symbol in a predefined symbol table. For example, you could select an instrument name from a hardware configuration table.

Note: Your ability to edit some parameters depends on the "SymVal" security setting for your user login name or group. See "Controlling System Security" in Chapter 6.

Output Designates parameters that will return results. You can designate only one output per action definition. (Use array parameters to pass multiple results.)

Note: You should only designate parameters of type Int32, Int32Array, Real64, Real64Array, String, or StringArray as Output because automatic limits checking is restricted to these types.

Restrict Value For some data items, specifies the low and high limits for permissible values for a data item or for all elements in an array.

Arrays Specifies arrays of up to three dimensions for Int32, Point, Range, Real64, and String data types. You can specify values, designate values by reference, set point values, or set range values for each element in an array.

To Add a Parameter to an Action

1. With an action definition loaded in the Action Definition Editor, choose the Add button at the bottom of the Action Definition Editor.

2. Specify the parameter's characteristics.

You can click a row under Value and choose the ... button to invoke a separate editor for the parameter or expand the tree view of the parameter’s characteristics and edit items directly.

See “Specifying the Properties for Parameters & Symbols” in Chapter 3 of the *Getting Started* book for general information about specifying parameters.

Note

If desired, more than one parameter in your action can return a result; i.e., more than one parameter can have its Action Output box checked in the editor used to specify that parameter’s characteristics. If more than one parameter returns a result, use the drop-down list to the right of “The current result is:” to specify which parameter’s result should be used for limits checking.

Tip: When working with a list of parameters, you can use the Move Up and Move Down buttons to reorder the list.

To Modify a Parameter to an Action

1. With an action definition loaded in the Action Definition Editor, click a parameter in the list under Action Parameters.

2. Modify the parameter’s characteristics as needed.

   You can click a row under Value and choose the ... button to invoke a separate editor for the parameter or expand the tree view of the parameter’s characteristics and edit items directly.

   See “Specifying the Properties for Parameters & Symbols” in Chapter 3 of the *Getting Started* book for general information about specifying parameters.

To Delete a Parameter to an Action

1. With an action definition loaded in the Action Definition Editor, click a parameter in the list under Action Parameters.

2. Choose the Delete button at the bottom of the Action Definition Editor.

Caution

If you delete a parameter and exit the Action Definition Editor without resaving the action definition, your change will be discarded.

Using Keywords with Actions

The next several topics describe the mechanics of working with keywords, which you associate with actions to make specific actions easier to find when searching libraries of actions.

To Add a Keyword to an Action

1. Do either of the following in the Action Definition window:

   - Select a keyword from the list of master keywords.

     - or -

   - If none of the predefined master keywords fits the action, type a new keyword in the keyword box. The new keyword should clearly identify the action's purpose. Typical keywords might be measure, setup, instrument, or stimulus.

Note You can assign multiple keywords for an action.

Note Use existing master keywords whenever possible. Adding too many new keywords can make it harder to find actions if the list of keywords becomes too long to browse conveniently.

2. Choose the Add button adjacent to the list of keywords.

To Delete a Keyword from an Action

1. In the Action Definition window, click on the keyword you want to delete from the list of keywords for the current action.

2. Choose the Delete button adjacent to the list of keywords.

To Add a Master Keyword to the List

1. Do either of the following in the Action Definition window:

    - Select a keyword from the list of keywords for the current action.

      - *or* -

    - Type a new keyword in the keyword box.

2. Choose Edit | Add Master Keyword in the menu bar.

Note

Minimize the number of master keywords that you add. Keywords are most useful when developers in an organization agree upon a standard, compact set of keywords whose meaning is specific.

To Delete a Master Keyword from the List

1. In the Action Definition window, select a keyword in the list of master keywords.

2. Choose Edit | Delete Master Keyword in the menu bar.

Creating Actions in C

Note Although most of this section describes using Visual C++ to create action code, *you do not need to know C++ to create actions*. Some topics describe C++ functionality for those who are familiar with C++, but in most cases you can simply follow the examples and make your code work. Typically, all you are doing is using a C++ compiler to produce C-like code; i.e., you are not using the C++ extensions to the C language. Thus, when you see reference to a “C action” it may help if you think of it as “C-like action code written using a C++ development environment.”

The action routines in a C action reside in a DLL whose code you write and compile. Each DLL can contain one or more action routines and, if desired, you can add new actions to an existing DLL.[1] You must decide how many or how few action routines to include in a single DLL.

What are the trade-offs? Using many small DLLs—for example, one DLL per action routine—causes testplans to load more slowly than having one large DLL that contains many action routines. However, using one large DLL reduces the modularity of your test system. We recommend using a single DLL to hold a logically related set of action routines, such as routines that make DC measurements.

Overview of the Process

The general process to follow when creating an action in C is:

1. Use HP TestExec SL’s Action Definition Editor to define the action by specifying the name of the routine (or names if using a setup/cleanup routine), parameters, descriptions, and keywords.

   For more information, see “To Define an Action.”

[1] The system DLLs supplied with HP TestExec SL are read-only, and you should not add new action routines to them.

2. Use your C/C++ environment to create a header file (“.h”) that declares the functions in your actions and an implementation file (“.cpp”) to contain the action source code.

Note

Be sure to enclose your action code in an `extern "C"` declaration, as shown in the examples, to prevent C++’s type-safe linkage scheme—i.e., “name mangling”—from causing problems when linking.

3. Write the code for the action routine.

4. Compile the source code to build a DLL.

5. Test/debug the DLL as needed.

Note

Your C/C++ environment does not need to be running while you use HP TestExec SL unless you are debugging an action and want to set breakpoints in the C/C++ environment.

Writing C Actions

C actions use the DLL action style, which passes named parameters in a block or collection that is a C structure. Instead of specifying each parameter in a formal list, you pass a handle to the entire parameter block. Unlike a formal list of parameters, individual parameters in a parameter block are referenced by name and not by position.

Note

Besides containing parameters used to pass values to action routines, parameter blocks also can contain parameters that return results used for limits checking. Thus, your C action code should not use an explicit “return” statement to return a value.

Using Parameter Blocks With a C Compiler

If you do not have a C++ compiler, you can use DLL style actions by using an API to access parameter values from a C compiler. The API provides a function for getting the value of each parameter type. The form for the function name is “UtaPbGet” or “UtaPbSet” plus the name of the parameter type. For example, the following lines of code declare a variable whose type is double and return a value to it from a parameter named `MyParm` in a parameter block.

```
double dMyVariable;
UtaPbGetReal64(hParmBlock, "MyParm", &dMyVariable);
```

Listed below are the various API functions and the types of parameters with which they are used.

This function name . . .	Gets/Sets a value for this parameter type
UtaPbGetComplex UtaPbSetComplex	Complex
UtaPbGetInst UtaPbSetInst	Inst
UtaPbGetInt32 UtaPbSetInt32	Int32
UtaPbGetInt32Array UtaPbSetInt32Array	Int32Array
UtaPbGetNode UtaPbSetNode	Node
UtaPbGetPath UtaPbSetPath	Path
UtaPbGetPoint UtaPbSetPoint	Point
UtaPbGetPointArray UtaPbSetPointArray	PointArray
UtaPbGetRange UtaPbSetRange	Range
UtaPbGetRangeArray UtaPbSetRangeArray	RangeArray
UtaPbGetReal64 UtaPbSetReal64	Real64

UtaPbGetReal64Array UtaPbSetReal64Array	Real64Array
UtaPbGetReal64Expr UtaPbSetReal64Expr	Real64Expr
UtaPbGetString UtaPbSetString	String
UtaPbGetStringArray UtaPbSetStringArray	StringArray

The following example shows how to use the API functions to access parameter blocks.

Note To understand the differences between using C and C++ compilers, you may find it useful to contrast this example with the similar example described later under “Using Parameter Blocks With a C++ Compiler.”

```
// C action routine to program a DVM & return a reading.
// Parameter block was defined with these parameters:
//    Result    - UtaReal64
//    Function  - UtaInt32
// Note: Use this routine with a C compiler.

#include <sicl.h>
#include <uta.h>
#define DEVICE_ADDRESS "hpib7,23"
{
void UTADLL read_dvm (HUTAPB hParmBlock)
  {
  long lDVM_Function;

  // HP TestCore API functions are used to return values
  // from the parameter block.
  UtaPbGetInt32(hParmBlock, "Function", &lDVM_Function);
double dRdg;
INST instID;
instID = iopen (DEVICE_ADDRESS);
iprintf (instID, "F%dRAN3T3\r\n" , lDVM_Function);
iscanf  (instID, "%lf\r\n" , &dRdg);
UtaPbSetReal64(hParmBlock, "Result", dRdg);
iclose (instID);
_siclcleanup();
}
```

For more information about the API functions used in the example, see the *Reference* book.

Using Parameter Blocks With a C++ Compiler

When you use a C++ compiler, HP TestExec SL's parameter types are defined as C++ classes that behave like ordinary C data types. This lets you write normal C code in action routines, except that variable declarations look slightly different.

When you declare a normal variable in C/C++, its declaration looks like this:

<*data type*> <*variable name*>;

An example of this is:

```
long lMyVariable;
```

The syntax used when declaring variables for parameter blocks with a C++ compiler in HP TestExec SL looks like this:

<*data type*> <*variable name*> (<*handle to parameter block*>, <*parameter name* or *ID*>);

The definition syntax lets you look up a parameter either by name or by ID. An example of this is:

```
IUtaInt32 lMyVariable(hParmBlock, "MyParameter")
```

Here, `IUtaInt32` is the C++ class that HP TestExec SL uses for long (32-bit integer) data. `lMyVariable` is the name of the variable being declared. `hParmBlock` is the handle to a parameter block that contains parameters being passed into the action routine. `MyParameter` is the name of a parameter (defined with the Action Definition Editor) in the parameter block whose value is to be passed to this variable.

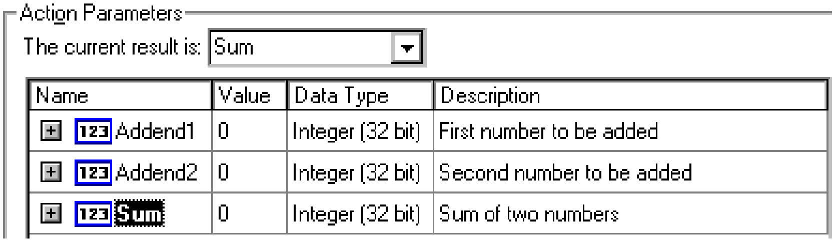
What about other data types? The list below shows the correspondence between the names of the C++ classes and the types of parameters supported by HP TestExec SL.

This C++ class. . .	Corresponds to this parameter type
IUtaComplex	Complex
IUtaInst	Inst
IUtaInt32	Int32
IUtaInt32Array	Int32Array
IUtaNode	(reserved)
IUtaPath	Path
IUtaPoint	Point
IUtaPointArray	(reserved)
IUtaRange	Range
IUtaRangeArray	(reserved)
IUtaReal64	Real64
IUtaReal64Array	Real64Array
IUtaReal64Expr	(reserved)
IUtaString	String
IUtaStringArray	StringArray
IUtaWaveform	Waveform

What might an example of using a parameter block look like? Suppose you used the Action Definition Editor to define an action whose parameter block contained these parameters:

Parameter Name	Parameter Type
Addend1	Int32
Addend2	Int32
Sum	Int32

As defined in the Action Definition Editor, the parameter block might look like this:



The current result is: Sum

Name	Value	Data Type	Description
⊞ 123 Addend1	0	Integer (32 bit)	First number to be added
⊞ 123 Addend2	0	Integer (32 bit)	Second number to be added
⊞ 123 Sum	0	Integer (32 bit)	Sum of two numbers

Inside an action routine written using a C++ compiler, you could then use the special declaration syntax to define three C variables corresponding to the parameters, like this:

```
IUtaInt32 lAddend1(hParmBlock, "Addend1");
IUtaInt32 lAddend2(hParmBlock, "Addend2");
IUtaInt32 lSum(hParmBlock, "Sum");
```

This declaration associates parameters in the parameter block with variables inside the action routine. After declaring the variables, you can use them as you would normal C variables of the corresponding type. For example, you can use `lAddend1`, `lAddend2`, and `lSum` as longs (32-bit integers).

Shown below is an excerpt from a simple action routine that uses the parameter block and variable declarations described above to add two integers and return their sum.

```
extern "C" { // Prevent C++ compiler from using name mangling
void UTADLL AddTwoIntegersExecute(HUTAPB hParmBlock)
  {
  // Declare local variables & associate them with parameters
  // in parameter block.
  IUtaInt32 lAddend1(hParmBlock, "Addend1");
  IUtaInt32 lAddend2(hParmBlock, "Addend2");
  IUtaInt32 lSum(hParmBlock, "Sum");

  // add the values together
  lSum = lAddend1 + lAddend2;
  }
}
```

Something important to note here is that the result, `Sum`, is returned via a parameter in the parameter block and not through a return data type—i.e., there is no explicit “return” statement that returns the result. When using parameter blocks, all passing of values between action code and HP TestExec SL is done via parameters in the block.

A more extensive example of using parameter blocks looks like this:

```
// C action routine to program a DVM & return a reading.
// Note: Use this routine with a C++ compiler.

#include <sicl.h>
#include <uta.h>
#define DEVICE_ADDRESS "hpib7,23"
extern "C"
{
void UTADLL read_dvm (HUTAPB hParmBlock)
  {
  // Use special syntax to declare two variables and associate them
  // with parameters in the parameter block.
  IUtaReal64 dDVM_Result(hParmBlock, "Result");
  IUtaInt32 lDVM_Function(hParmBlock, "Function");
```

```
   INST instID;
   instID = iopen (DEVICE_ADDRESS);
   iprintf (instID, "F%dRAN3T3\r\n" , lDVM_Function);
   iscanf  (instID, "%lf\r\n" , &dDVM_Result);
   iclose (instID);
   _siclcleanup();
   }
}
```

The parameter block for the example above looks like this:

Parameter Name	Parameter Type
Result	Real64
Function	Int32

The special syntax for declaring C variables corresponding to the parameters in the example looks like this:

```
IUtaReal64 dDVM_Result(hParmBlock, "Result");
IUtaInt32 lDVM_Function(hParmBlock, "Function");
```

Given these declarations, you can use `dDVM_Result` like a double and `lDVM_Function` like a long (32-bit integer).

Some of the C++ classes directly correspond to standard C data types, as shown below.

This C++ class . . .	Corresponds to this C data type
IUtaInt32	int or long (whichever is 32 bits)
IUtaInt32Array	int[] or long[] (whichever is 32 bits)
IUtaReal64	double
IUtaReal64Array	double[]
IUtaString	const char*

Note The Real64Expr parameter type is treated as a Real64 type.

If the HP TestExec SL parameter type does *not* readily correspond to a C data type, the argument passed to the user routine behaves as if it is of type HUTADATA. To access the value of an argument passed as this type, your action routine must use special access routines defined for that particular data type.

The following list shows the correspondence between C++ classes, parameter types in HP TestExec SL, and HUTA data types.

C++ Class	Parameter Type	HUTA Data Type
IUtaComplex	Complex	HUTACOMPLEX
IUtaInst	Inst	HUTAINST
IUtaPath	Path	HUTAPATH
IUtaPoint	Point	HUTAPOINT
IUtaPointArray	PointArray	HUTAPOINTARRAY
IUtaRange	Range	HUTARANGE
IUtaRangeArray	RangeArray	HUTARANGEARRAY
IUtaStringArray	StringArray	HUTASTRINGARRAY
IUtaWaveform	Waveform	HUTAWAVEFORM

A set of API functions let you access these types of parameters. Also, additional functions are provided for directly accessing the handles to the various data types. For more information about these data types, the API calls used with them, and how to read the syntax of the data types and APIs, see the *Reference* book.

Exception Handling in C Actions

Note

For an overview of exceptions, see “About Exceptions” in the *Getting Started* book.

Various functions in the Exception Handling API let you use C actions to raise and examine exceptions that occur during testing. Because it lets you handle exceptions at a low level, handling exceptions in actions can be more precise than simply letting your testplan branch to an alternate sequence of tests in the Exception Sequence.

The remainder of this topic shows some of the most useful concepts and functions in the Exception Handling API. You can find a full list of these API functions and their descriptions in Chapter 4 of the *Reference* book.

At the simplest level, the `UtaExcRaiseUserError()` function lets you raise a user-defined exception in response to some anticipated error condition. The example below shows how this works.

```
// Example causes the following to display in Report window when
// encountered while testplan is executing:
//   Condition raised a user-defined exception! (Severity: 5)

char chMessage [60];
long lSeverity;
...(do something)
if (some condition == some value) // raise an exception?
  {
  strcpy (chMessage, "Condition raised a user-defined exception!");
  lSeverity = 5;
  UtaExcRaiseUserError(chMessage, lSeverity);
  }
...(testing is aborted because exception occurred)
```

As noted in the example’s comments, unless they are specifically received and handled otherwise, user-defined exceptions simply send a message to the Report window and abort testing.

However, much of the power in having user-defined exceptions lies in being able to process them and act appropriately instead of simply aborting testing the first time an exception occurs. The next example uses several of the Exception Handling API functions in a more meaningful way.

```
// Example can raise user-defined exceptions while action is doing
// tasks. Each exception has a severity level associated with it. Near
// the end of the action, a routine checks to see if exceptions
// occurred and receives them if they did. If the severity of an
// exception exceeds a threshold, a value of -1 is written to a
// parameter named "parm1" in the action's parameter block. If "parm1"
// is a reference to a symbol in a symbol table, actions in other tests
// can access the symbol table to see if this action raised one or more
// "serious" exceptions.

HUTAEXC hUtaException;
long lSeverity, lNumExceptions, lCounter;
char chMessage[40];
...(do something)
// action routine contains one or more routines to see if an
// exception condition exists

if (some condition == some value) // raise an exception?
  {
  strcpy (chMessage, "Condition raised an exception!");
  lSeverity = 10; // assign severity level to this exception
  UtaExcRaiseUserError(chMessage, lSeverity); // raise exception
  }

...(testing continues)
...
...(near end of testing)
if (UtaExcRegIsError()) // if exception(s) exist
  {
  lNumExceptions = UtaExcRegGetErrorCount(); // get # of exceptions
  // receive all exceptions & get handle to first in list
  hUtaException = UtaExcRegReceiveError();
  for (lCounter = 1; lCounter <= lNumExceptions; lCounter++)
    {
    if (UtaExcGetSeverity(hUtaException) > 5) // test severity
      UtaPbSetInt32(hParmBlock, "parm1", -1); // write to parm.
    if (lCounter < lNumExceptions)
      // get handle to next exception
      hUtaException = UtaExcGetNextError(hUtaException);
    };
  };
```

In a similar fashion, you can raise, receive, and handle user-defined exceptions specific to your testing environment. You have the choice of handling exceptions immediately or, as in the example, postponing their handling until later.

The API functions used the preceding example are listed below.

This function...	Does this...
UtaExcRaiseUserError()	Raises a user-defined exception and lets you specify an error message and severity indicator to be associated with the exception.
UtaExcRegIsError()	Tests for the presence of one or more exceptions that have been raised but not yet received.
UtaExcRegGetErrorCount()	Returns the number of exceptions that have been raised and not yet received.
UtaExcRegReceiveError()	Returns a handle to the first in a list of exceptions.
UtaExcGetSeverity()	Returns the severity level that was set when the exception occurred.
UtaExcGetNextError()	Given the handle to an exception, returns the handle to the next exception if more than one exception has been raised.

HP TestExec SL also has predefined exceptions for such conditions as math errors, out-of-range values, and array dimensioning errors. These are listed in the system file “uta.h”.

Using C Actions to Control Switching Paths

Overview

If you do not use hardware handler software to communicate with switching modules, you must control switching directly from actions via your chosen I/O strategy. This requires you to write custom routines that tend to be complex and may not be reusable.

But if you are using a hardware handler, there are two better ways to control switching during a test:

- You can use the Test Executive’s Switching Path Editor to graphically control switching paths at the beginning and end of a test. This is the easiest method.
- You can use API functions to control switching paths from an action, which lets you modify switching paths during a test. Because it requires you to write code, this method is more difficult to use than the Switching Path Editor. However, it is more versatile because it lets you explicitly control switching as needed.

When action routines contain code that controls switching paths, they tend to be specific to a particular implementation of switching hardware. This can make them more specialized and less reusable than action routines that do not control switching. In general, you can improve the reusability of actions by specifying switching at the test level instead of inside action routines.

Using API Functions to Control Switching Pathss

The C Action Development API contains the following functions you can use to control switching from a C action routine.

API Function	Purpose
UtaPathConnect()	Establishes a switching path
UtaPathDisconnect()	Resets all relays in a switching path
UtaPathWait()	Waits for the switching relays to close before returning

The declaration of the `UtaPathConnect()` function is:

```
void UtaPathConnect (HUTAPATH hPath, BOOL bWait=TRUE);
```

`UtaPathConnect()` establishes the switching path specified by `hPath`, which is most likely passed into the action routine as one of its parameters. The `bWait` parameter is optional; it defaults to TRUE, but if set to FALSE the function will return without waiting for relays to close.

An example using `UtaPathConnect()` might look like this:

```
UtaPathConnect (hPath);
```

The syntax of the `UtaPathDisconnect()` function is:

```
void UtaPathDisconnect (
  HUTAPATH hPath,
  BOOL bWait = TRUE
  );
```

`UtaPathDisconnect()` resets all the relays along the path. Reset is defined by the default positions of the switching elements. Thus, the path is opened. The `bWait` parameter is optional; it defaults to TRUE, but if set to FALSE the function will return without waiting for relays to open.

An example using `UtaPathDisconnect()` might look like this:

```
UtaPathDisconnect (hPath);
```

The `UtaPathWait()` function provides a way to tell the system to wait for a specific path connection. Its syntax is:

```
void UtaPathWait (HUTAPATH hPath);
```

An example that includes `UtaPathWait()` might look like this:

```
UtaPathConnect (hPath, FALSE);
...(do something else while waiting)
// Ensures that path will be closed.
UtaPathWait (hPath);
```

Notice that unlike using `UtaPathConnect()` by itself with `bWait` set to TRUE, having a `UtaPathWait()` function follow a `UtaPathConnect()` whose value for `bWait` is FALSE lets you do other tasks while waiting for the specified switching path, `hPath`, to be established by `UtaPathConnect()`.

HP TestExec SL also provides a `UtaPbGetPath()` function you can use to retrieve switching path data from parameter blocks and subsequently use with the functions described above. An example of its use looks like this:

```
HUTAPATH hPath;
// Get the parameter specifying the path
hPath = UtaPbGetPath (hParameterBlock, "DcvPathLow");

// Close the path
UtaPathConnect (hPath);

// Take a measurement
// ... code that takes a measurement

// Open the Path
UtaPathDisconnect (hPath);
```

For more information, see “Functions for Manipulating Switching Paths from Actions” in Chapter 2 of the *Reference* book.

Using States to Store Switching Data

Using the `UtaPathConnect()` and `UtaPathDisconnect()` functions to control switching paths is convenient in simple cases, but requires more work in more complex situations. For example, suppose you need to set up new paths temporarily and restore them later. This could cause you to write quite a few lines of code to track the changing states of switching elements in switching paths.

To remedy this, the C Action Development API provides various “UtaState...” functions used to create and manipulate “switching states” that contain one or more switching paths. You can find a complete list of them and their syntaxes under “Functions for Manipulating Switching Paths from Actions” in Chapter 2 of the *Reference* book.

Consider the following example, which temporarily stores the state of the switching hardware, adds to the state of the switching hardware a path previously stored as “NewPath” in a parameter block, and subsequently restores the switching hardware to its original state.

```
HUTASTATE hOriginalState; // variable for handle to switching state
HUTAPATH hPath; // variable for handle to switching path
hOriginalState = UtaStateCreate(); // create empty switching state
hPath = UtaPbGetPath(hParameterBlock, "NewPath"); // get path data
UtaStateMergePathState(hOriginalState, hPath); // define state’s scope
UtaStateUpdate(hOriginalState); // store current state of hardware
UtaPathConnect(hPath); // set hardware to path retrieved from NewPath
// Do tasks while new path is in effect
...
...(make a measurement, etc.)
...
// restore the hardware to its initial, stored state
UtaStateRecall(hOriginalState);
UtaStateRelease(hOriginalState); // free memory used by state object
```

How does the example work? Suppose we begin by using `UtaStateCreate()` to create a switching state:

```
hOriginalState = UtaStateCreate();
```

In theory, an empty (uninitialized) switching state potentially could store switching information for an entire test system. However, in reality it is quicker and more convenient to work with a subset of all possible switching hardware. A switching path defines just such a subset, so the next line gets the data associated with an existing switching path stored in a parameter in a parameter block, like this:

```
hPath = UtaPbGetPath(hParameterBlock, "NewPath");
```

Now we must merge the switching path data with the empty switching state to define the scope of the switching state; i.e., which specific hardware out of all possible hardware it describes. To do this, we use `UtaStateMergePathState()`, as shown below.

```
UtaStateMergePathState(hOriginalState, hPath);
```

Now the range of the switching state is restricted to the path specified by the data retrieved from the parameter named “NewPath”. Note that this data may not describe the exact state of the switching elements that we need; i.e., relays or other programmable connections defined in the switching path may be in the wrong positions for our intended task. However, that causes no problem because *merging a path into a state changes nothing in the actual hardware*. Instead, the purpose of merging is simply to define the scope or extent of the switching state.

Next we want to store the current status of the hardware—i.e., the positions of the switching elements in the path of interest—before changing it. Storing the hardware's status in a switching state lets us store and recall it as a single entity, instead of laboriously manipulating it via individual `UtaPathConnect()` and `UtaPathDisconnect()` statements. We use `UtaStateUpdate()` to update the state from the current settings of the hardware:

```
UtaStateUpdate(hOriginalState);
```

Having safely stored the state of the switching hardware, we can use the "NewPath" data to change it, like this:

```
UtaPathConnect(hPath);
```

After doing tasks that require the new switching path, we can restore the original path in a single statement with `UtaStateRecall()`:

```
UtaStateRecall(hOriginalState);
```

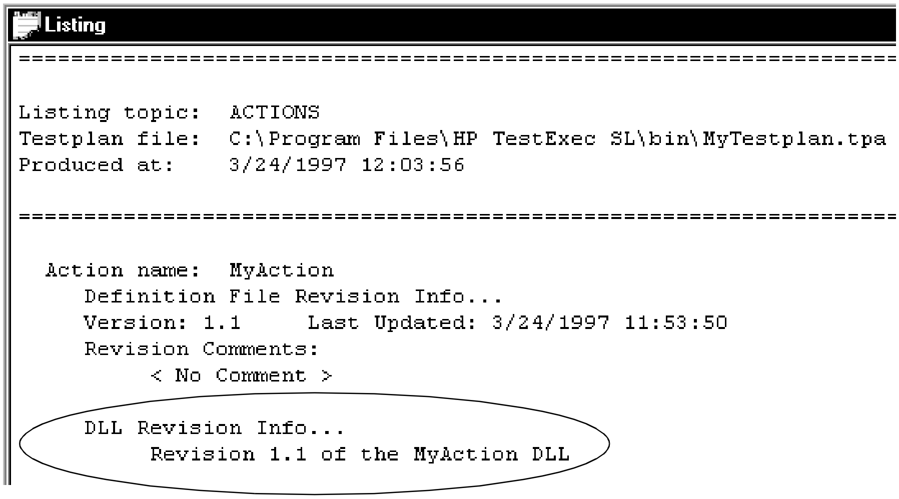
Now that we have returned the hardware to its original state, we are finished using the state object created at the beginning of the example. To free the memory it is using, we do this:

```
UtaStateRelease(hOriginalState);
```

For more information about switching states, see "Data Types Associated with Switching" in Chapter 1 of the *Reference* book.

Adding Revision Control Information for Actions

If desired, each DLL in which action code resides can include a string of text for auditing purposes, such as revision control information. As shown below, the text appears when listing the actions in a testplan.



```
=====================================================================

Listing topic:  ACTIONS
Testplan file:  C:\Program Files\HP TestExec SL\bin\MyTestplan.tpa
Produced at:    3/24/1997 12:03:56

=====================================================================

  Action name:  MyAction
     Definition File Revision Info...
     Version: 1.1     Last Updated: 3/24/1997 11:53:50
     Revision Comments:
          < No Comment >

     DLL Revision Info...
          Revision 1.1 of the MyAction DLL
```

The method for doing this is shown in the example of action code below. You must add a macro named UTA_DECLARE_DLL_REVISION_TEXT and specify the auditing text in it. Be sure to place the macro inside the scope of the declaration for `extern "C"` when using a C++ compiler.

```
// File "MyAction.cpp"
#include "stdafx.h"
#include <uta.h>
#include "MyAction.h"

extern "C" { // Prevent C++ compiler from using name mangling

// All actions in this DLL share the following auditing information
UTA_DECLARE_DLL_REVISION_TEXT ("Your auditing text goes here...");

void UTADLL MyActionRoutine (HUTAPB hParmBlock)
   {
   ...(code that implements the action routine)
   return;
   }
```

```
    ...(additional action routines implemented in this DLL)

  } // end extern "C"
```

Note

You can add *one string of auditing text per DLL*. That string of text appears in the listing for all actions implemented in the DLL. If you use multiple source files for your DLL, be sure to specify the auditing text in only one of them or you will generate an error.

Example of Creating a C Action in a New DLL

Note

The topics in this section describe how to use the development environment provided with Microsoft Visual C++. If you are using another C/C++ development environment, the details will vary but the concepts will be similar.

This section, which assumes you are somewhat familiar with the mechanics of using Visual C++, describes how to create a new action in a new DLL. The emphasis is on the process of creating an action in C, not on what belongs inside an action.

Defining the Action

Follow the general procedure described earlier in “Defining an Action” and keep the following in mind when using the Action Definition Editor to define actions written in C:

- Choose “DLL Style” as the action style.
- The executable code for the action must reside in a DLL. Enter the name of that DLL as the library name for the action.
- You can define execute, setup, or setup/cleanup routines for C actions.
- For the Routine name, use the name of the C routine.
- Be sure to use parameter types that are appropriate for the C language.

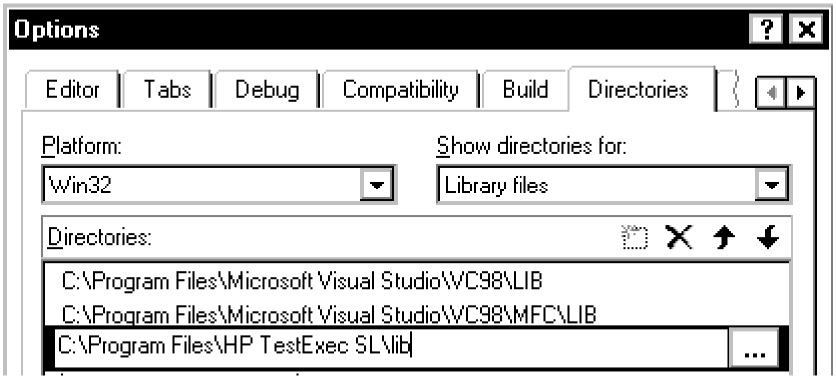
Specifying the Development Environment Options

You set the Visual C++ development environment options once, and then they become the defaults for any new projects that you create.

Specifying the Path for Libraries

1. Choose Tools | Options in the Visual C++ menu bar.

2. In the Options box, choose the Directories tab and specify a path for library files that includes the “lib” directory beneath the home directory in which HP TestExec SL is installed on your system. An example is shown below.



Platform: Win32

Show directories for: Library files

Directories:

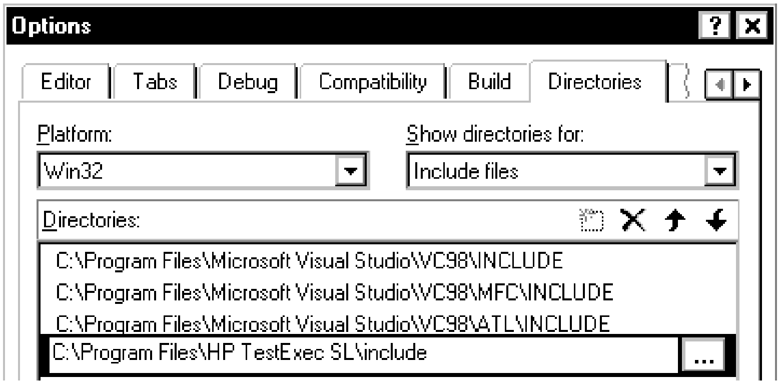
C:\Program Files\Microsoft Visual Studio\VC98\LIB
C:\Program Files\Microsoft Visual Studio\VC98\MFC\LIB
C:\Program Files\HP TestExec SL\lib

Note

Depending upon where you installed Visual C++ and HP TestExec SL on your system, your paths may vary from those shown.

Specifying the Path for Include Files

1. In the Options box, specify a path for include files that includes the “include” directory beneath the home directory in which HP TestExec SL is installed on your system. An example is shown below.



Platform: Win32
Show directories for: Include files
Directories:
C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\ATL\INCLUDE
C:\Program Files\HP TestExec SL\include

2. Click the OK button to save the path you specified.

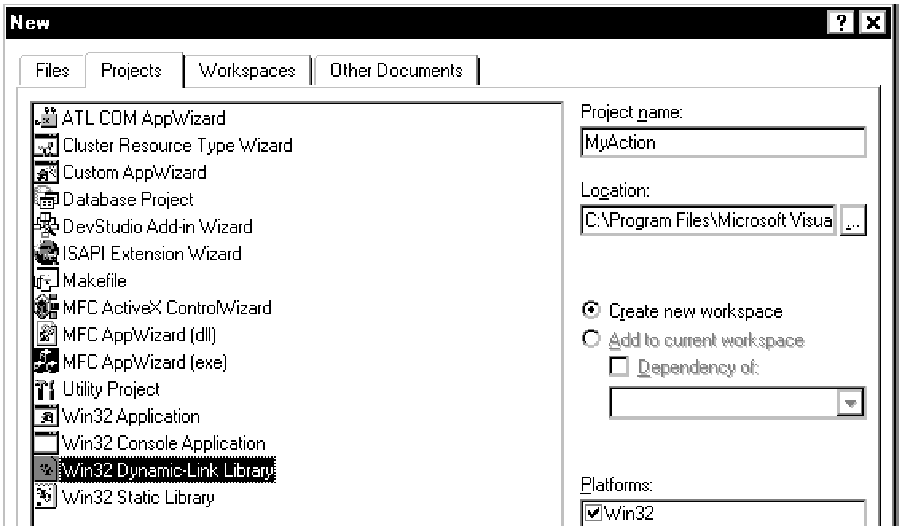
Note

Depending upon where you installed Visual C++ and HP TestExec SL on your system, your paths may vary from those shown.

Creating a New DLL Project

1. Choose File | New in the Visual C++ menu bar.

2. Choose the Projects tab and specify Win32 Dynamic-Link Library as the type of project, as shown below.



Project name: MyAction

Location: C:\Program Files\Microsoft Visua

● Create new workspace

○ Add to current workspace

□ Dependency of:

Platforms:

☑ Win32

3. Type a Name for your project.

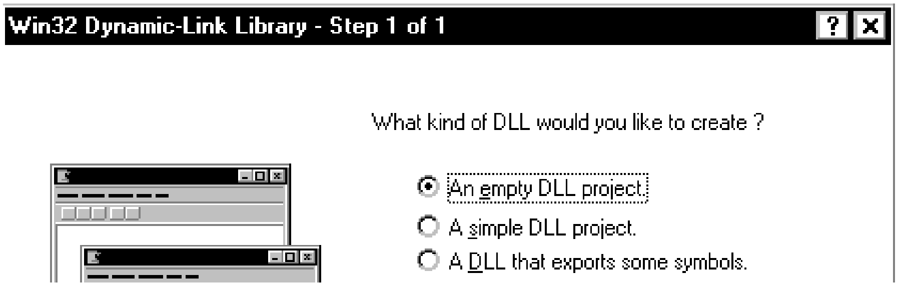
4. Specify the Location for your project.

Note

The action definition created with HP TestExec SL’s Action Definition Editor needs to reference this location. If you later recompile the DLL in release mode and move it elsewhere, you need to specify its new location as described in “Specifying the Search Path for Libraries” in Chapter 5.

5. Choose the OK button.

6. Choose to create an empty DLL project, as shown below.



What kind of DLL would you like to create ?

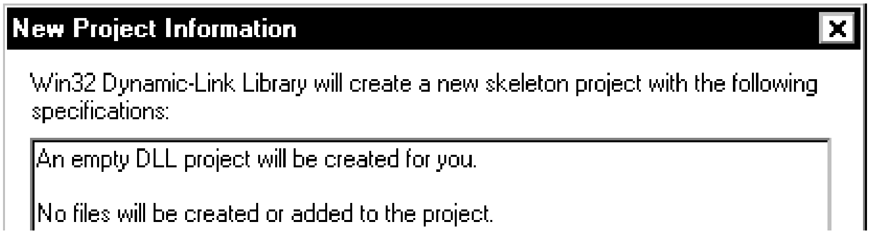
● An empty DLL project.

○ A simple DLL project.

○ A DLL that exports some symbols.

7. Choose the Finish button.

8. Verify the information for the new project, as shown below.



Win32 Dynamic-Link Library will create a new skeleton project with the following specifications:

An empty DLL project will be created for you.

No files will be created or added to the project.

9. Choose the OK button.

Specifying the Project Settings

You specify the project settings once for each new project you create.

1. Choose Project | Settings in the Visual C++ menu bar.

2. If needed, choose the General tab to make its options visible.

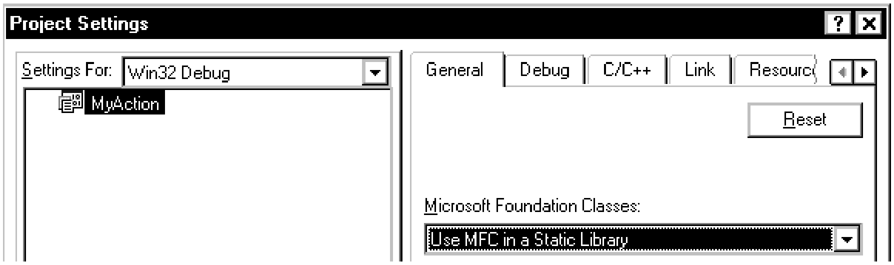
3. In the Project Settings box, specify the Microsoft Foundation Classes (MFC) option.

Specify this...	If you wish to do this...
Not Using MFC	Create a DLL that is small and fast but does not support MFC’s features.[a] This option is most useful for reducing overhead when you have many individual action routines in many DLLs.
Use MFC in a Static Library[b]	Use MFC’s features but have a large DLL. Because the size of your action code typically will be far smaller than the DLL’s overhead, this option is most useful when you have only a few DLLs and each of them contains multiple, related action routines.

[a] Generally speaking, MFC’s most useful feature insofar as actions are concerned is that it lets you use visual resources, such as dialog boxes, in actions. In many cases these graphical features are not needed to manipulate data or control instruments, and you do not need to use MFC.

[b] You should not use the MFC in a Shared Dll option because HP TestExec SL already does this, and having different versions of MFC may cause conflicts.

An example of specifying “Use MFC in a Static Library” for the Microsoft Foundation Classes option is shown below.



Settings For: Win32 Debug

Microsoft Foundation Classes: Use MFC in a Static Library

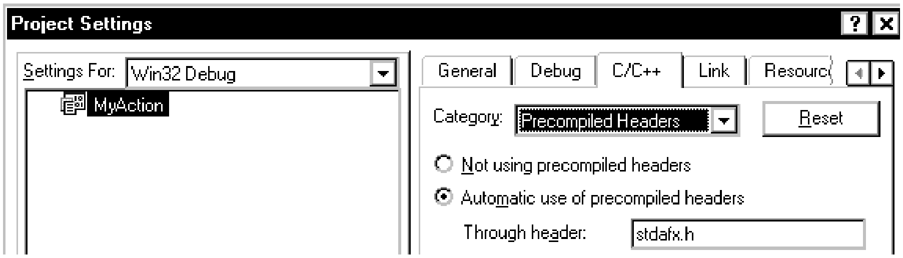
4. Choose the C/C++ tab to make its options visible.

5. Choose Precompiled Headers from the Category list.

6. Be sure “Automatic use of precompiled headers” is enabled.

7. Specify “stdafx.h” for the “Through header” option.

   When you specify precompiled headers, the compiler will compile once all the header files through the one specified in the dialog box, and after that it will compile only your code. This speeds subsequent compilations. An example of using these options is shown below.



Project Settings

Settings For: Win32 Debug

MyAction

Category: Precompiled Headers

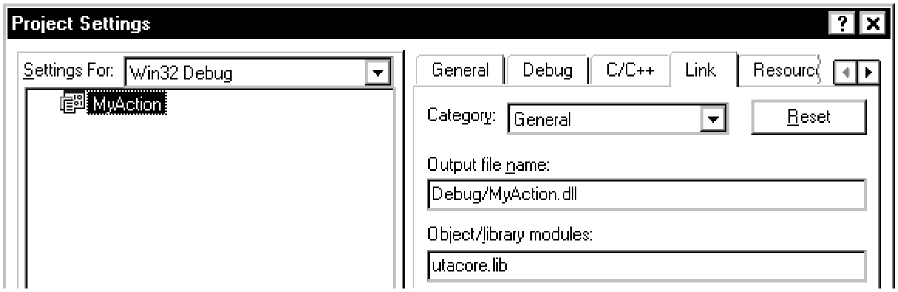
○ Not using precompiled headers

● Automatic use of precompiled headers

Through header: stdafx.h

8. Choose the Link tab to make its options visible.

9. Specify “utacore.lib” for the “Object/Library modules” option, as shown below.



Project Settings

Settings For: Win32 Debug

MyAction

Category: General

Output file name: Debug/MyAction.dll

Object/library modules: utacore.lib

   Linking against “utacore.lib” lets the compiler resolve all the external references to HP TestCore definitions, functions, and classes used in your action code. Because you already specified the default library path earlier, you do not need to enter the full path here.

10. Choose the OK button to save the project settings and close the Project Settings box.

Writing Source Files for the Action Code

There are a couple of ways to write action code. You may prefer to write the code from scratch, or you can copy the code for an existing action and use it as a template for a new action. Shown below are the contents of the sample files needed to create a simple action from scratch. Put the files in the project directory for your DLL.

Contents of the Header File:

```
// This file is MyAction.h
extern "C" void UTAAPI MyExecuteFunction(HUTAPB hParameterBlock);
```

Contents of the Implementation File:

```
// This file is MyAction.cpp
#include "stdafx.h"
#include <uta.h> // API for HP TestCore services
#include "MyAction.h"

CWinApp theApp; // Comment out or remove this line if not using MFC
extern "C"{ // Prevent C++ compiler from using name mangling
void UTADLL MyExecuteFunction(HUTAPB hParameterBlock)
  {
  // Action code to do a task goes here...
  return;
  }
}
```

Contents of the System-Level Include File:

```
// This file is stdafx.h
#define VC_EXTRALEAN // Exclude rarely used stuff
#include <afxwin.h> // MFC core and standard components
#include <afxext.h> // MFC extensions
```

Although this code is used to create a DLL that contains a single execute action, you could write multiple actions of various types and put them all in a single DLL. Also, your actions typically will use parameters passed in a parameter block.

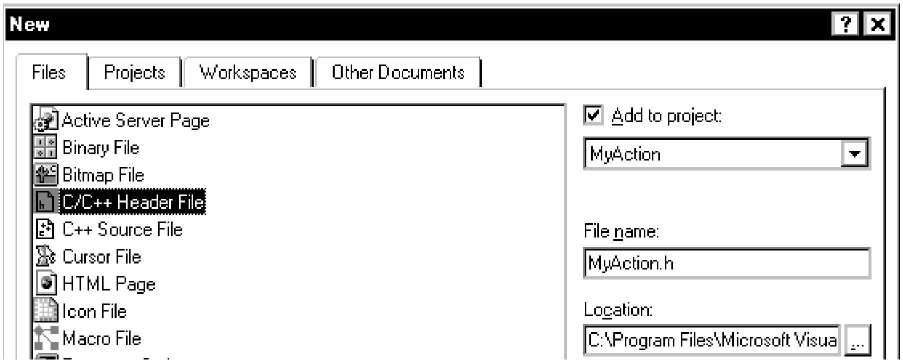
Adding Source Files to the Project

Do the following for each of the source files above:

1. Choose File | New in the Visual C++ menu bar.

2. On the Files tab in the New box, specify the file's type, name[1], and location, and choose the OK button to add it to your project.

   An example of creating a header file is shown below.



☑ Add to project: MyAction

File name: MyAction.h

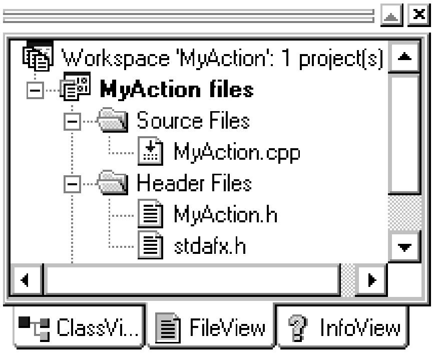
Location: C:\Program Files\Microsoft Visua

3. Type the file's contents in the editor window that appears.

[1] Use a ".cpp" extension for your implementation file.

Verifying the Project’s Contents

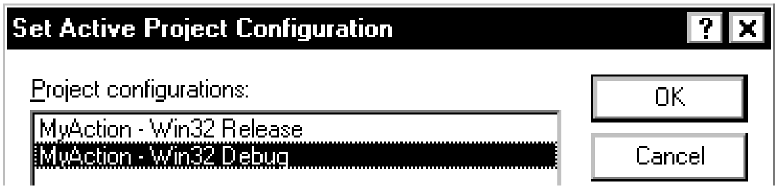
- Choose the FileView pane in the Visual C++ workspace window to verify the contents of your project, as shown below.


Workspace 'MyAction': 1 project(s)
MyAction files
Source Files
MyAction.cpp
Header Files
MyAction.h
stdafx.h

Choosing Which Configuration to Build

1. Choose Build | Set Active Configuration... in the Visual C++ menu bar.

2. Specify that you wish to build a debug version of the project, as shown below.


Set Active Project Configuration

Project configurations:
MyAction - Win32 Release
MyAction - Win32 Debug

Note The debug version of a program contains additional code that makes it larger and slower to execute than a release version. Thus, you probably will want to build a final, release version of the DLL after you have debugged it.

3. Choose the OK button.

Building the Project

- Choose Build | Build <*project name*> in the Visual C++ menu bar to build the DLL.

Copying the DLL to Its Destination Directory

Overview

Each time you create a DLL containing action routines, you need to copy the DLL to the destination directory where it will be used. You can greatly simplify and reduce potential errors in the copying process by creating one or more custom tools in Visual C++.

Note Any time you move a DLL, you potentially need to need to specify its new location as described in “Specifying the Search Path for Libraries” in Chapter 5. Also, if you are running a testplan, you need to close and reopen it before new or moved files will take effect.

Creating a Custom Tool to Copy the DLL

1. Choose Tools | Customize in Visual C++’s menu bar.

2. In the Customize box, choose the Tools tab.

3. Choose the New icon that appears above the “Menu contents” list.

4. In the blank field that just appeared in the “Menu contents” list, specify a descriptive label for what this tool does.

5. Type `xcopy` as the Command.

6. Click the arrow to the right of the Arguments field.

7. Choose Target Path from the list that appears.

8. Enter quotes around the `$(TargetPath)` entry in the Arguments field.

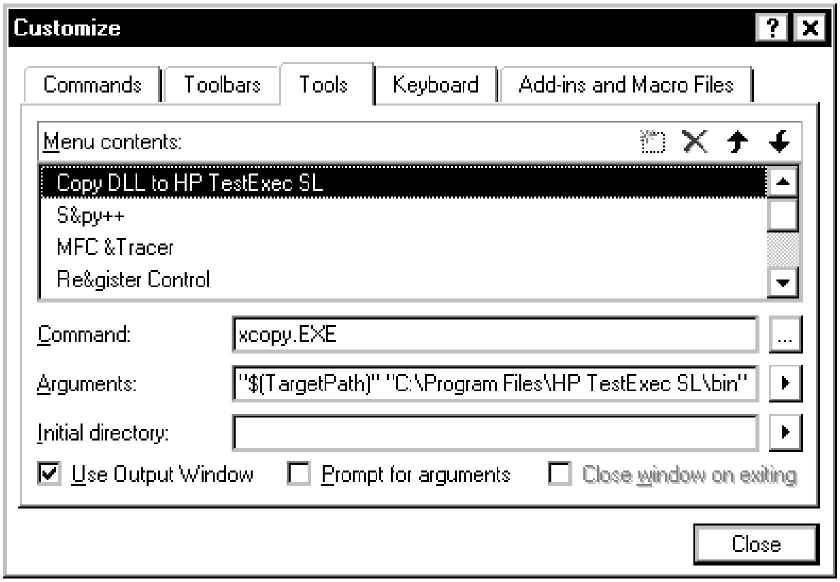
9. To the right of `"$(TargetPath)"` in the list of Arguments, type the name of the destination directory to which your DLL should be copied.

   Tip: If your pathname includes spaces, be sure to enclose it in quotes.

10. Enable the check box labeled Use Output Window.

11. Choose the Close button.

Shown below is an example of specifying the options for this custom tool.



Command: xcopy.EXE

Arguments: "$(TargetPath)" "C:\Program Files\HP TestExec SL\bin"

Initial directory:

☑ Use Output Window □ Prompt for arguments □ Close window on exiting

Using the Custom Tool to Copy the DLL

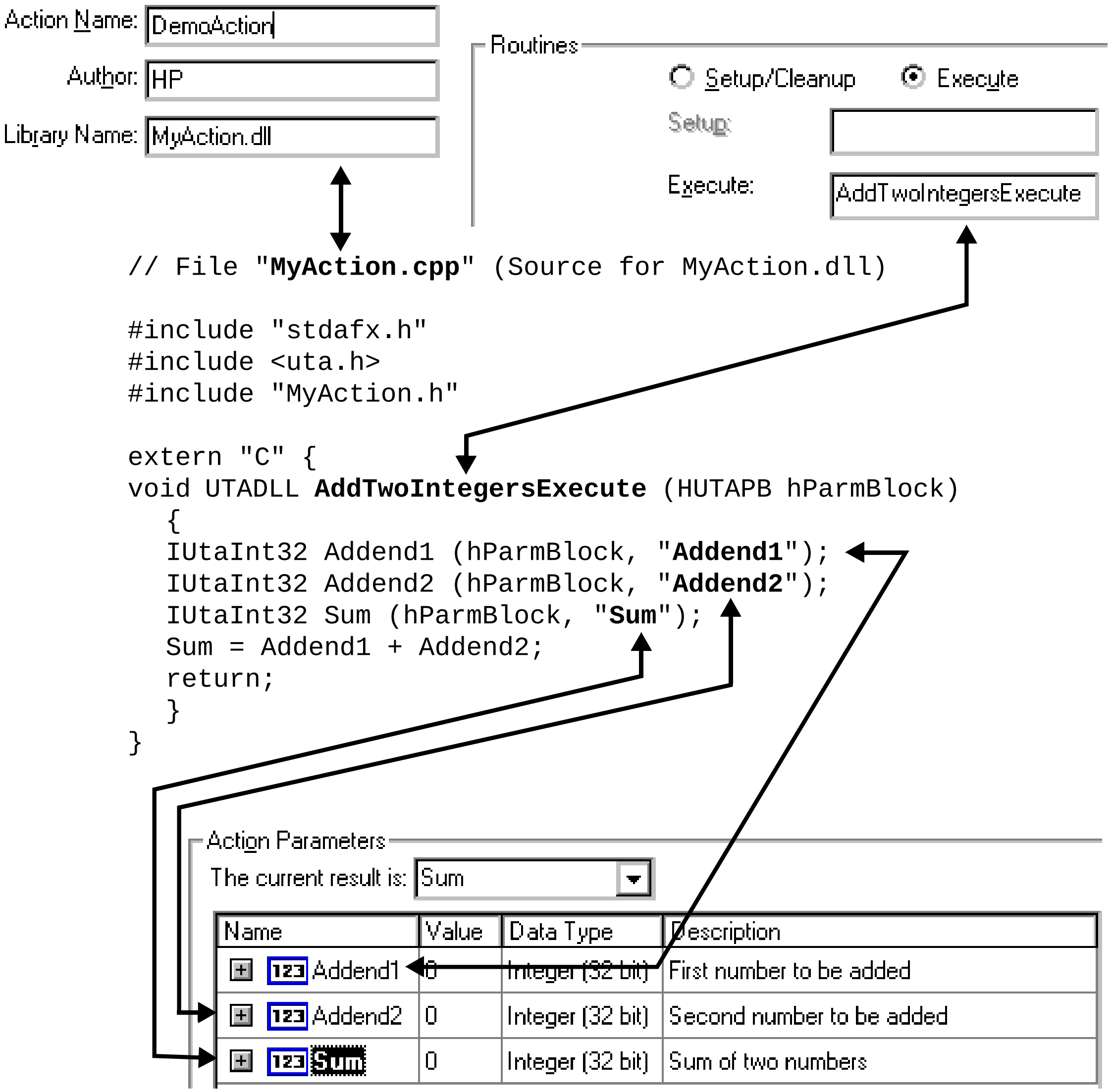
1. Choose Tools in Visual C++’s Tools menu bar.

2. Choose the custom tool from the menu of tools.

When the tool runs, its results appear in Visual C++’s output window.

Example of Defining a C Action

The illustration below shows how information you specify in the Action Definition Editor relates to the associated code in a C action routine.



Action Name: DemoAction

Author: HP

Library Name: MyAction.dll

Routines

○ Setup/Cleanup ● Execute

Setup:

Execute: AddTwoIntegersExecute

```
// File "MyAction.cpp" (Source for MyAction.dll)

#include "stdafx.h"
#include <uta.h>
#include "MyAction.h"

extern "C" {
void UTADLL AddTwoIntegersExecute (HUTAPB hParmBlock)
   {
   IUtaInt32 Addend1 (hParmBlock, "Addend1");
   IUtaInt32 Addend2 (hParmBlock, "Addend2");
   IUtaInt32 Sum (hParmBlock, "Sum");
   Sum = Addend1 + Addend2;
   return;
   }
}
```

Action Parameters

The current result is: Sum

Name	Value	Data Type	Description
⊞ 123 Addend1	0	Integer (32 bit)	First number to be added
⊞ 123 Addend2	0	Integer (32 bit)	Second number to be added
⊞ 123 Sum	0	Integer (32 bit)	Sum of two numbers

Adding a C Action to an Existing DLL

Follow this general procedure to add a new C action to an existing DLL:

1. Use HP TestExec SL's Action Definition Editor to create a definition for the new action. When you specify the library, use the name of the existing ".dll" file to which you want to add the new action.

2. If the existing DLL was created using a different compiler or different compiler options, verify that your C/C++ development environment's options are similar to those described earlier in "Specifying the C/C++ Development Environment Options."

3. Use your development environment to open the project workspace or make (".mak") file—i.e., whichever method your C/C++ development environment uses to manage projects—for the existing DLL.

4. Add the code for the new action to the implementation (".cpp") file.

   The example below shows the code for an implementation file used to create a DLL that contains two action routines. Notice that the declaration for `extern "C"` encompasses both functions, and that the implementation file uses the UTADLL macro in the functions.

```
// File is "MyAction.cpp"
#include "stdafx.h"
#include <uta.h>
#include "MyAction.h"

extern "C" { // Prevent C++ name mangling

  // Function that adds two integers
  void UTADLL AddTwoIntegersExecute (HUTAPB hParmBlock)
    {
    IUtaInt32 Addend1 (hParmBlock, "Addend1");
    IUtaInt32 Addend2 (hParmBlock, "Addend2");
    IUtaInt32 Sum (hParmBlock, "Sum");
    Sum = Addend1 + Addend2;
    return;
    }
```

```
   // Function that adds two reals
   void UTADLL AddTwoRealsExecute (HUTAPB hParmBlock)
     {
     IUtaReal64 Addend1 (hParmBlock, "Addend1");
     IUtaReal64 Addend2 (hParmBlock, "Addend2");
     IUtaReal64 Sum (hParmBlock, "Sum");
     Sum = Addend1 + Addend2;
     return;
     }

} // end extern "C"
```

5. Add the declaration for the new action to the header (“.h”) file.

   The example below shows the code for a header file that contains the prototypes for two action routines. Notice that each prototype includes a declaration for `extern "C"`, and that the header file uses the UTAAPI macro in the prototypes.

```
//File is "MyAction.h"
extern "C" void UTAAPI AddTwoIntegersExecute(HUTAPB hParmBlock);
extern "C" void UTAAPI AddTwoRealsExecute(HUTAPB hParmBlock);
```

6. Rebuild the DLL.

   Tip: You can use the “dumpbin” utility provided with Visual C++ to browse the contents of an existing DLL. The example below shows an excerpt from a “dumpbin /exports” listing that shows the exported names of the functions in a DLL.

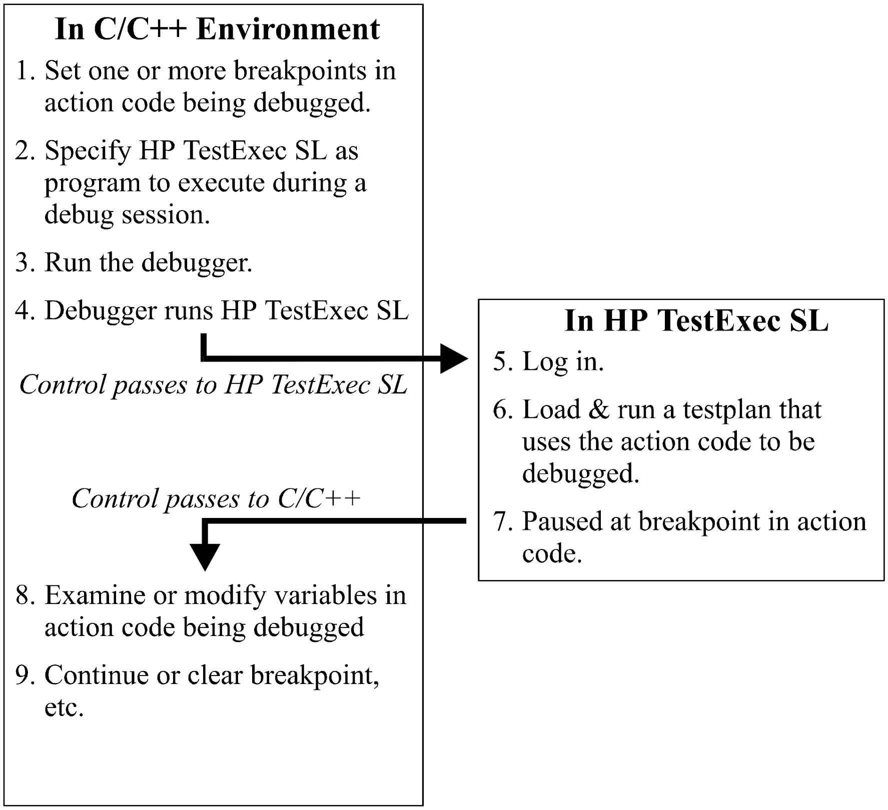
```
ordinal hint   name
      1    0   _DisplayExceptions@4  (00001680)
      2    1   _EchoInt32@4  (00001040)
      3    2   _EchoReal64@4  (000010A0)
      4    3   _RandomFailReal64@4  (000012B0)
```

Note

Any time you move a DLL, you potentially need to specify its new location as described in “Specifying the Search Path for Libraries” in Chapter 5. Also, if you are running a testplan, you need to close and reopen it before new or moved files will take effect.

Debugging C Actions

You can debug C actions with the debugging tools provided by the C/C++ environment in which you program. The general sequence of events when using your C/C++ environment for debugging is:



In C/C++ Environment

1. Set one or more breakpoints in action code being debugged.
2. Specify HP TestExec SL as program to execute during a debug session.
3. Run the debugger.
4. Debugger runs HP TestExec SL

Control passes to HP TestExec SL

In HP TestExec SL

5. Log in.
6. Load & run a testplan that uses the action code to be debugged.
7. Paused at breakpoint in action code.

Control passes to C/C++

8. Examine or modify variables in action code being debugged
9. Continue or clear breakpoint, etc.

Note Debugging may require that you build a new DLL specifically for debug purposes.

Note Any time you move a DLL, you potentially need to specify its new location as described in “Specifying the Search Path for Libraries” in Chapter 5. Also, if you are running a testplan, you need to close and reopen it before new or moved files will take effect.

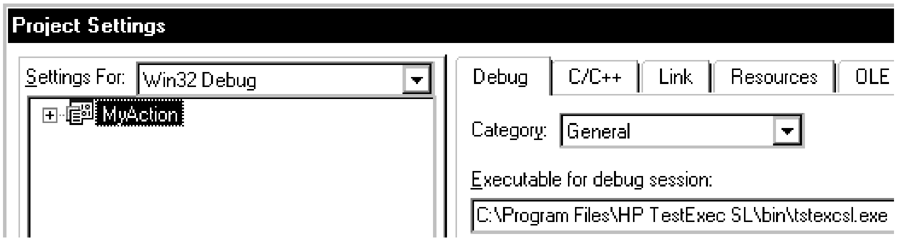
Note If your debug process causes you to modify and recompile a DLL that contains action code, you cannot simply copy the modified DLL over the existing DLL while HP TestExec SL has a testplan loaded that uses that DLL. Instead, you must close the testplan, copy the modified DLL over the existing DLL, and then reload the testplan.

Follow this general procedure to debug a C action:

1. Run your C/C++ development environment.

2. Specify "*<HP TestExec SL home>*\bin\tstexcsl.exe" as the program to support debug.

   An example of doing this in Visual C++ is shown below.



Project Settings

Settings For: Win32 Debug

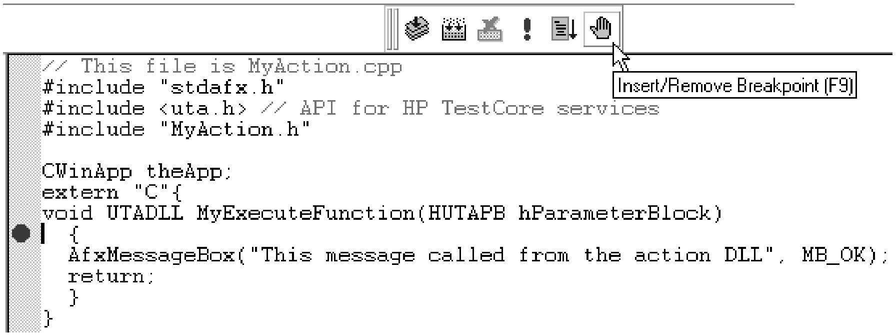
MyAction

Category: General

Executable for debug session: C:\Program Files\HP TestExec SL\bin\tstexcsl.exe

3. Set the desired breakpoints in the implementation file (".cpp") for the action.

   An example of doing this in Visual C++ is shown below.



```
// This file is MyAction.cpp
#include "stdafx.h"
#include <uta.h> // API for HP TestCore services
#include "MyAction.h"

CWinApp theApp;
extern "C"{
void UTADLL MyExecuteFunction(HUTAPB hParameterBlock)
  {
  AfxMessageBox("This message called from the action DLL", MB_OK);
  return;
  }
}
```

4. Choose whichever button or command runs your debugger.

In Visual C++, choose Build | Start Debug | Go in the menu bar.

5. After HP TestExec SL has loaded, load or create a testplan that invokes the action being debugged.

6. Run the testplan.

7. When the breakpoint in your action code is reached and control is returned to the debugger, use the debugger’s features to debug the action.

Another useful debugging technique is to create action code that pops up a message box or dialog box and stops test execution so you can use external instruments to diagnose problems.

Creating Actions in HP VEE

HP TestExec SL lets you write actions in HP VEE and take advantage of HP VEE's features, such as debugging and instrument control. Executable HP VEE actions are HP VEE user functions stored in an HP VEE library.

Creating an action in HP VEE is a two-step process. You can do the following steps in any order:

- Use HP VEE to create the functions used by the action, and save the resulting user functions in the HP VEE library.
- Use the Action Definition Editor to define the action so the Test Executive is aware of its characteristics.

See also: "HP VEE Considerations" in Chapter 2 of the *Getting Started* book.

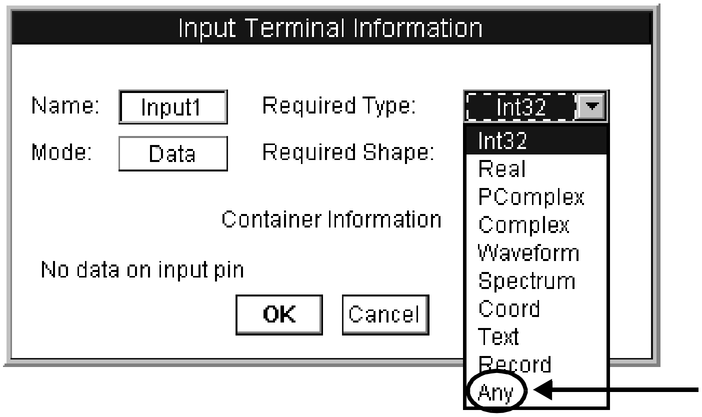
Restrictions on Parameter Usage in HP VEE

HP VEE only lets you pass certain types of parameters. Shown below is a list of those types and how they correspond to one another in both environments.

In HP TestExec SL	**In HP VEE**	
Data Type:	*Data Type:*	*Shape:*
Int32	Int32	Scalar
Int32Array	Int32	Array
Real64	Real	Scalar
Real64Array	Real	Array
String	Text	Scalar
StringArray	Text	Array

Note

As shown below, you must explicitly specify a data type for pins in UserFunctions; i.e., do not use the Any type.



Input Terminal Information

Name: Input1

Mode: Data

Required Type: Int32

Required Shape:

Container Information

No data on input pin

Do not use this option

Defining an HP VEE Action

Be aware of the following when using the Action Definition Editor to create HP VEE action definitions:

- You must choose “HP VEE” as the action style.
- When defining the action library name, enter the name of the HP VEE library—e.g., “mylib.vee”—that contains the user function that does the action.
- For the Routine name, enter the name of the HP VEE function; i.e., the user function in the specified HP VEE library.

Example of an HP VEE Action

This section provides a simple example of how parameters are passed between HP TestExec SL and action code created using HP VEE. The action is done by an HP VEE user function that receives two parameters from HP TestExec SL, generates a random number based on those parameters, and then passes the result back to HP TestExec SL.

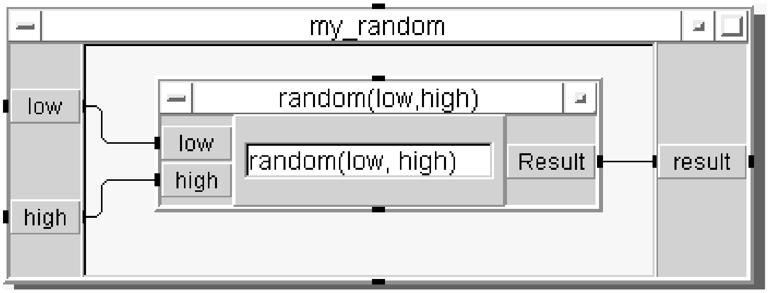
All that is required to pass parameters between HP TestExec SL and HP VEE is to:

- Make the names of parameters in the HP VEE user function match the names of corresponding parameters specified for the action in the Action Definition Editor.
- Make the name of the HP VEE function match the name of the action code specified in the Action Definition Editor.

Suppose you have used the Action Definition Editor to provide the following action definition information and stored it in a file called “random.umd” located in an action library where you chose to store HP VEE action definitions.

Action name	random
Description	Generates a random number.
Library name	c:\project\vee\mylib.vee
Routine name	my_random
Parameters	(All parameters are of type Real.)
low	The low range value for the random number generator.
high	The high range value for the random number generator.
result	The resulting value from the HP VEE random number generator (designated as an OUTPUT in the Action Definition Editor).

The corresponding HP VEE user object used to create the user function for this definition might look like this:



Debugging HP VEE Actions

In a production environment, you probably want HP TestExec SL to schedule HP VEE in run-only mode. However, this means that none of HP VEE's command menus are present, which prevents you from setting breakpoints, editing files, starting or stopping programs, or controlling HP VEE in any way.

The Action Definition Editor provides an option that helps you debug HP VEE actions. If you click to select the Debug check box in the Action Definition window, HP VEE will be run in debug mode. After you have debugged your actions, unselect the box to return to run-only mode.

Tip: While debugging HP VEE actions, you can edit UserFunctions by running another copy of HP VEE and making edits there. After editing a UserFunction, be sure to save the changes to disk with File | Save in HP VEE. Then close and reopen the current testplan in HP TestExec SL to load the changes.

Error Handling in HP VEE

If you select the Debug check box in the Action Definition Editor's Action Definition window, errors in HP VEE will not cause exceptions. Instead, the normal HP VEE processing will handle the error. An error message dialog box will appear, giving the complete text of the error message and highlighting in red the HP VEE object containing the error.

Controlling the Geometry of HP VEE Windows

If desired, you can specify the geometry for the window in which HP VEE actions appear. Use a text editor, such as WordPad in its text mode, to add two lines in the following format to the "tstexcsl.ini" file in HP TestExec SL's home directory (which by default is "\Program Files\HP TestExec SL").[1]

```
[VEE Actions]
Geometry=WidthxHeight+XOffset+YOffset
```

All dimensions are measured in pixels.

The example below specifies a window that is 800 pixels wide, 500 pixels high, and originates in the upper-left corner of the screen.

```
[VEE Actions]
Geometry=800x500+0+0
```

Executing HP VEE Actions on a Remote System

If desired, you can execute HP VEE actions on a host system other than the one on which you are running HP TestExec SL. Use a text editor, such as WordPad in its text mode, to add two lines in the following format to the "tstexcsl.ini" file in HP TestExec SL's home directory (which by default is "\Program Files\HP TestExec SL").[2]

```
[VEE Actions]
HostName=RemoteActionHost
```

where *RemoteActionHost* is the domain name or IP address of a remote system where HP VEE is installed. For example,

```
[VEE Actions]
HostName=hplvlfl.lvd.hp.com
```

or

```
[VEE Actions]
HostName=15.11.89.216
```

[1] If the [VEE Actions] section already exists, simply add the missing line to it.

[2] If the [VEE Actions] section already exists, simply add the missing line to it.

Note

An action executing on a remote system appears in a window on the remote system.

Creating Actions in National Instruments LabVIEW

HP TestExec SL lets you write actions in National Instruments LabVIEW and take advantage of National Instruments LabVIEW's features, such as debugging and instrument control. Executable code for National Instruments LabVIEW actions is National Instruments LabVIEW virtual instruments (VIs) stored in a National Instruments LabVIEW library (".llb") file.

Creating an action in National Instruments LabVIEW is a two-step process. You can do the following steps in any order:

- Use National Instruments LabVIEW to create the VIs used by the action, and save the resulting routines in the library.
- Use the Action Definition Editor to define the action so the Test Executive is aware of its characteristics.

Related Files

HP TestExec SL includes the following National Instruments LabVIEW-related files:

uta.llb Contains predefined VIs for passing parameters to and from HP TestExec SL. Located in directory "\\<*HP TestExec SL home*>\libs".

We suggest that you place this library in a subdirectory called "uta.lib" in the National Instruments LabVIEW installation directory. If you do not want to create a subdirectory of that name, install the library in another subdirectory of the National Instruments LabVIEW installation directory and make sure the directory has a ".lib" extension.

utaactn.llb Contains a VI used to ask National Instruments LabVIEW to execute specific VIs for HP TestExec SL. Note that the front panel of this VI occupies a small amount of space on your monitor's screen. Located in directory "\\<*HP TestExec SL home*>\bin".

Restrictions on Parameter Passing

Be aware of the following restrictions when passing parameters between HP TestExec SL and National Instruments LabVIEW:

- You can only pass certain types of parameters. Shown below is a list of those types and how they correspond to one another in both environments.

In HP TestExec SL	In National Instruments LabVIEW
Data Type:	*Data Type:*
Int32	Signed 32-bit integer
Int32Array	Array of signed 32-bit integers
Real64	Eight-byte double precision number

Real64Array	Array of eight-byte double precision numbers
String	C string
StringArray	Array of C strings

- You must use a VI to pass parameters between the two environments.

  Custom VIs provided by Hewlett-Packard let you make a graphical connection between parameters in HP TestExec SL and standard VIs used with National Instruments LabVIEW. Parameters are passed in a named block or group.

  To access an HP TestExec SL parameter, place one of the VIs from library “uta.llb” in the diagram of the action’s VI. If the library was installed correctly, you can select VIs in it by choosing Functions | UTA in National Instruments LabVIEW.

List of Custom VIs Provided with HP TestExec SL

The functionality of VIs that pass parameters is viewed from the perspective of the National Instruments LabVIEW environment. Thus, the names of VIs that send a value to HP TestExec SL contain the word “set” and the names of VIs that retrieve a value from HP TestExec SL contain the word “get.”

The custom VIs provided with HP TestExec SL that support the passing of parameters are:

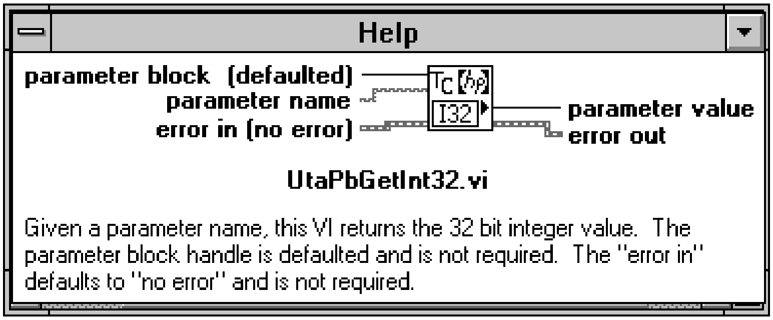
UtaPbGetInt32.vi	Obtains the value of Int32 parameters.
UtaPbSetInt32.vi	Updates Int32 parameters with new values.
UtaPbGetInt32Array.vi	Obtains the value of Int32Array parameters.
UtaPbSetInt32Array.vi	Updates Int32Array parameters with new values.
UtaPbGetReal64.vi	Obtains the value of Real64 parameters.
UtaPbSetReal64.vi	Updates Real64 parameters with new values.
UtaPbGetReal64Array.vi	Obtains the value of Real64Array parameters.

UtaPbSetReal64Array.vi	Updates Real64Array parameters with new values.
UtaPbGetString.vi	Obtains the value of String parameters.
UtaPbSetString.vi	Updates String parameters with new values.
UtaPbGetStringArray.vi	Obtains the value of StringArray parameters.
UtaPbSetStringArray.vi	Updates StringArray parameters with new values.

An additional VI is provided that lets you use National Instruments LabVIEW to control a switching path:

UtaPathConnectNodes.vi	Connects nodes in a switching path. Useful if you need to modify a switching path within an action.

As with other VIs used with National Instruments LabVIEW, these custom VIs have front panels and onscreen help you can browse to learn more about them. An example of help for `UtaPbGetInt32.vi` shown below.



Given a parameter name, this VI returns the 32 bit integer value. The parameter block handle is defaulted and is not required. The "error in" defaults to "no error" and is not required.

Defining a National Instruments LabVIEW Action

Be aware of the following when using the Action Definition Editor to define National Instruments LabVIEW actions:

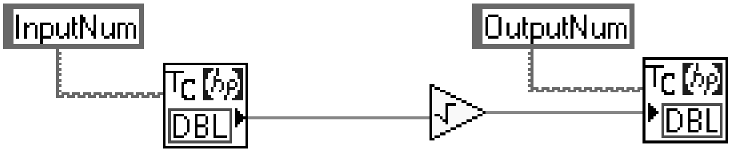
- You must choose “LabVIEW” as the action style.

- For the Library name, enter the name of the National Instruments LabVIEW VI library, including its “.llb” extension.

- For the Routine name, enter the National Instruments LabVIEW VI name, including its “.vi” extension; i.e., the action VI in the specified library.

Example of a National Instruments LabVIEW Action

Shown below is a simple example of a VI created using National Instruments LabVIEW with two of the custom VIs provided with HP TestExec SL.



The example shows how custom VIs are used to pass parameters between HP TestExec SL and National Instruments LabVIEW. Here, the custom VI named `UtaPbGetReal64` gets a parameter from the HP TestExec SL environment. The output from `UtaPbGetReal64` is connected to the input of the standard National Instruments LabVIEW VI used to take the square root of a number. The resulting square root is connected to the custom `UtaPbSetReal64` VI, which passes the result back to HP TestExec SL.

Shown below is the information you would use the Action Definition Editor to specify for this example. Notice how the names of parameters in the action definition match the names of the parameters of each library VI.

Action name	lvsqrt
Description	Takes the square root of a number.
Library name	c:\labview\cmlib.lib
Routine name	sqrt.vi

Parameters (All parameters are of type Real64.)

InputNum The number to be passed to National Instruments LabVIEW whose square root is to be taken.

OutputNum A parameter to hold the square root of the input number (designated as an OUTPUT in the Action Definition Editor).

The action definition is stored in a file called “lvsqrt.umd” located in a standard library for National Instruments LabVIEW action definitions.

Setting Interface Options for National Instruments LabVIEW

When HP TestExec SL executes a National Instruments LabVIEW action, the front panel of the VI associated with the action is displayed while the VI executes. This lets a test operator use the panel.

You can control the size and location of this panel. When HP TestExec SL executes the action, the panel window appears at the location and size you set when developing the action. If the action does not require any interaction with the test operator, you can make the panel size very small and place the panel in an inconspicuous part of the screen. This prevents the operator from being distracted by the panel.

You can also control which menus and toolbars display with the panel window, how the panel window looks, and numerous other options. Set these options by choosing the “Window Options” mode of the “VI Setup” dialog box in National Instruments LabVIEW.

Creating Actions in HP BASIC for Windows

HP TestExec SL lets you write actions in HP BASIC for Windows and take advantage of your familiarity with that instrument control language. Executable HP BASIC for Windows actions are SUB programs you write and add to a program that runs HP BASIC for Windows as a server for HP TestExec SL. Besides containing SUB programs that implement actions, the server program loads the graphical I/O environment (HP BASIC Plus), does any desired autostart configuration tasks, and runs the IPC Widget[1] that lets HP BASIC for Windows and HP TestExec SL communicate.

Creating an action in HP BASIC for Windows is a multi-step process. You can do the following steps in any order.

- Use HP BASIC for Windows to append one or more SUB programs containing your action code to a copy of the server template in file “server.prg”. This creates your HP BASIC for Windows server program.
- Use the Action Definition Editor to define the action so the Test Executive is aware of its characteristics.
- Use the HP BASIC for Windows “rmb_conf.exe” utility to “register” your server program and define its communications characteristics.

Note When HP TestExec SL calls an action written in HP BASIC for Windows, it automatically loads and runs HP BASIC for Windows.

[1] In HP BASIC for Windows, a “widget” is an entity created on the screen with an ASSIGN statement from an executing HP BASIC Plus program.

Related Files

HP TestExec SL includes the following HP BASIC for Windows-related files:

rmb_conf.exe	A utility used to define the characteristics of your HP BASIC for Windows server program.
server.prg	An HP BASIC for Windows program file for use as a template when creating your HP BASIC for Windows server program.
wiipc.dll	The IPC Widget used by the HP BASIC for Windows server program.
wiipc.hlp	A help file for the IPC Widget.
widgcom.dll	A helper DLL for the IPC Widget.
widgcom.csb	An HP BASIC for Windows CSUB used by the HP BASIC for Windows server program.

HP TestExec SL installs these files in the home directory in which HP BASIC for Windows is installed.

Restrictions on Parameter Usage in HP BASIC for Windows

HP BASIC for Windows only lets you pass certain types of parameters. Shown below is a list of those types and how they correspond to one another in both environments.

In HP TestExec SL	In HP BASIC for Windows
Data Type:	*Data Type:*
Int32	INTEGER (16-bit)
Int32Array	INTEGER Array
Real64	REAL
Real64Array	REAL Array

Complex	COMPLEX
String	String
StringArray	String Array

Note Integers are 32-bit in HP TestExec SL and 16-bit in HP BASIC for Windows. If you pass Int32 or Int32Array data to an HP BASIC for Windows action, be sure to restrict the value to a 16-bit range; i.e., -32768 through +32767. If you need values outside this range, use Real64 types instead of Int32.

Defining an HP BASIC for Windows Action

Be aware of the following when using the Action Definition Editor to create HP BASIC for Windows action definitions:

- You must choose “HP RMB” as the action style.
- Leave the Library Name field blank.
- For the Routine name, enter the name of the HP BASIC for Windows subprogram; i.e., the name of a SUB in the HP BASIC for Windows server program.

Creating an HP BASIC for Windows Server Program

Action code you write in HP BASIC for Windows resides in a server program that you create from a template provided by Hewlett-Packard. Do the following to create the server program:

1. Start HP BASIC for Windows if it is not already running.
2. Copy the server template (“server.prg”) to a new name, which will be the name of the server program that contains the action code you write. For example,

   ```
   copy "server.prg" to "MyServer.prg"
   ```

3. On the HP BASIC for Windows command line, load the renamed server template. For example,

   ```
   load "MyServer.prg"
   ```

4. Type “edit” on the command line and press Enter to begin editing your server program.

5. Add code that implements one or more actions. Begin adding your new code on a new line beyond the end of the existing program.

   Action code follows the general form shown below (line numbers have been omitted).

   ```
   ...(existing code in server template)
   SUB <name of action routine>
   COM /<name of action routine>/ <data type> <parameter name>
   ...
   ...(code that does a task suitable for an action)
   ...
   SUBEND
   ```

   Notice that the name of the action routine must be the same in the SUB and COM statements. Each action routine must have a unique name and its code must reside within its own matching pair of SUB and SUBEND statements. If you need to pass more than one data type in parameters to your action specified in a COM statement, use spaces between each data type and its first parameter, and commas as delimiters elsewhere, like this:

   ```
   <data type 1> <parm>,<parm>,<data type 2> <parm>,<parm>
   ```

   Keep the following in mind when writing actions:

   - Place all the actions for any given testplan in a single server program.

   - Do not use the STOP statement. It will cause the server to disconnect from HP TestExec SL.

   - Use ON ERROR and ON TIMEOUT trapping where appropriate to avoid a paused—i.e., “hung”—call to an action.

- We recommend that you do not use ON...RECOVER unless you have a thorough understanding of program flow when using a server.
- Remember that HP TestExec SL waits for your SUB to complete and return. Thus, if you use PAUSE or DIALOG statements, the user must interact with HP BASIC for Windows instead of with HP TestExec SL to restore testplan flow. But if HP BASIC for Windows is iconified, the user will be unaware that interaction is required. Either be sure users know when interaction is required or add GESCAPE CRT,32 at the beginning of interactive SUBS to keep them from being iconified.

6. Save the edited server template. For example,

   ```
   re-store "MyServer.prg"
   ```

7. Run the server configuration utility (“rmb_conf.exe”), specify the name of your modified server template in its Server Program field, and choose the OK button to save the change and exit.

Note Do not use spaces in pathnames in the configuration utility. Instead, use short pathnames as they appear in a DOS shell window. For example, instead of typing “c:\Program Files\HPBASIC” you must type “c:\Progra~1\HPBASIC”.

Note Unless you have all of your HP BASIC for Windows actions in a single server program, you must rerun the server configuration utility and specify the name of the appropriate server program each time you change testplans. If you change testplans often, you may want to add the server configuration utility to the Tools menu, as described under “Adding Custom Tools to HP TestExec SL” in Chapter 6.

Note Actions execute the fastest when HP BASIC for Windows is iconified. You can use the Start As option in the “rmb_conf.exe” utility to specify whether your server program starts as an icon or a window.

Example of an HP BASIC for Windows Action

This section provides a simple example of how parameters are passed between HP TestExec SL and action code created using HP BASIC for Windows. The action is done by an HP BASIC for Windows SUB program that receives one parameter—a radius—from HP TestExec SL, generates the diameter and area of a circle based on that parameter, and then passes the results back to HP TestExec SL via two other parameters.

All that is required to pass parameters between HP TestExec SL and HP BASIC for Windows is to do the following in your server program:

- Make the name of the HP BASIC for Windows SUB program match the name of the action routine specified in the Action Definition Editor.
- Create a labeled COM block with a name that matches the SUB name.
- List the parameters in the COM block in the same order as they appear in the Action Definition Editor.

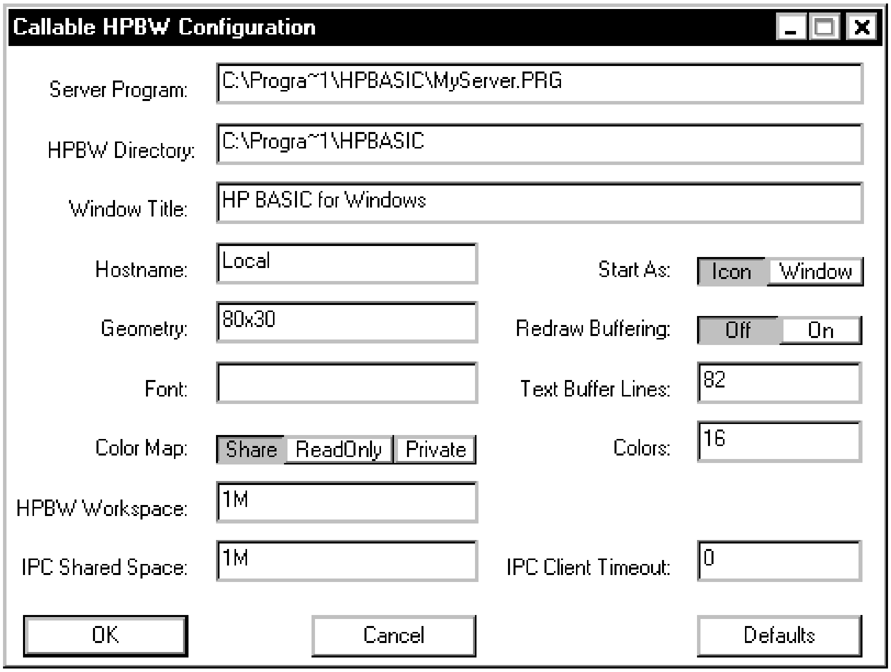
Suppose you have used the Action Definition Editor to provide the following action definition information and stored it in a file called “circle.umd” located with your HP BASIC for Windows actions.

Action name	Circle
Description	Calculates the diameter and area of a circle from its radius.
Library name	(*none*)
Routine name	Circle_math
Parameters	(All parameters are of type REAL.)
Radius	The specified radius of the circle.
Diameter	The calculated diameter of the circle (designated as an Output).
Area	The calculated area of the circle (designated as an OUTPUT in the Action Definition Editor).

The corresponding HP BASIC for Windows SUB used to implement the action might look like this (line numbers have been omitted):

```
...(existing code in server program)
SUB Circle_math
COM /Circle_math/ REAL Radius,Diameter,Area
!
Diameter=2*Radius
Area=PI*(Radius^2)
SUBEND
```

And the corresponding configuration for the server program might look like this:



Callable HPBW Configuration

Server Program: C:\Progra~1\HPBASIC\MyServer.PRG

HPBW Directory: C:\Progra~1\HPBASIC

Window Title: HP BASIC for Windows

Hostname: Local

Start As: [✓ Icon] Window

Geometry: 80x30

Redraw Buffering: [✓ Off] On

Font:

Text Buffer Lines: 82

Color Map: [✓ Share] ReadOnly Private

Colors: 16

HPBW Workspace: 1M

IPC Shared Space: 1M

IPC Client Timeout: 0

Debugging HP BASIC for Windows Actions

You can use standard features of the interactive HP BASIC for Windows environment when debugging actions. For example, you can pause, single-step, interrogate or modify the values of variables, list program segments, and use various debugging features provided by HP BASIC for

Windows. Also, keep the following in mind when debugging HP BASIC for Windows actions:

- Although the performance of actions created in HP BASIC for Windows actions is best when HP BASIC for Windows is iconified, interactive debugging requires a normal—i.e., non-iconified—window. The Start As option in the “rmb_conf.exe” utility lets you specify whether your HP BASIC for Windows server program starts as a window or an icon. Once started, you can use standard mouse interaction in Windows to maximize, minimize, or move the window.

- While interacting with HP BASIC for Windows, do not STOP or RESET the program because a stopped server disconnects from its HP TestExec SL client. You can use PAUSE, STEP, and CONTINUE.

- If you plan to interact with your HP BASIC for Windows workspace, we strongly recommend that you leave the value of IPC Client Timeout at 0 (zero) in the “rmb_conf.exe” utility. Otherwise, a paused action will eventually generate a timeout error.

4

Working with Switching Topology

This chapter describes how to use switching topology, which is a combination of physical and logical descriptions that define the switching configuration and interconnections between resources and the unit under test.

For an overview of switching topology, see Chapter 3 in the *Getting Started* book.

Defining the Switching Topology

When you “define” switching topology, you describe its characteristics so the Test Executive is aware of switchable paths in your test system. Also, you make the Test Executive aware of hardware modules that are available as resources during testing.

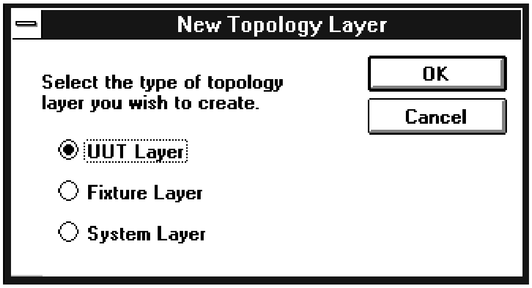
Note Your overall goal in defining the switching topology is to describe the hardware well enough to let the Switching Path Editor control switching paths during a test, but not so well that you describe every nuance of how the test system is wired. Thus, your emphasis should be on describing switching paths inside modules and any wires that interconnect these switching paths.

Overview

The Switching Topology Editor lets you define the three layers of topology for your test system. This topology information resides in three files:

<*system_name*>.ust	Contains a definition of the system layer.
<*fixture_name*>.ust	Contains a definition of the fixture layer.
<*UUT_name*>.ust	Contains a definition of the UUT layer.

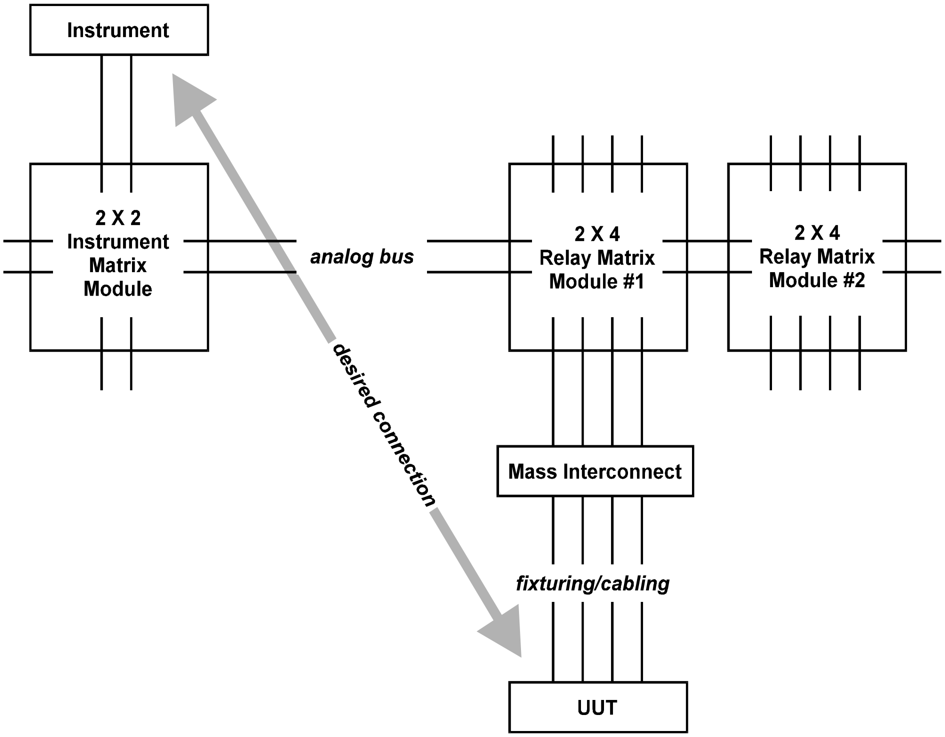
When you specify which layer to create in the Switching Topology Editor, it loads the appropriate file.


New Topology Layer

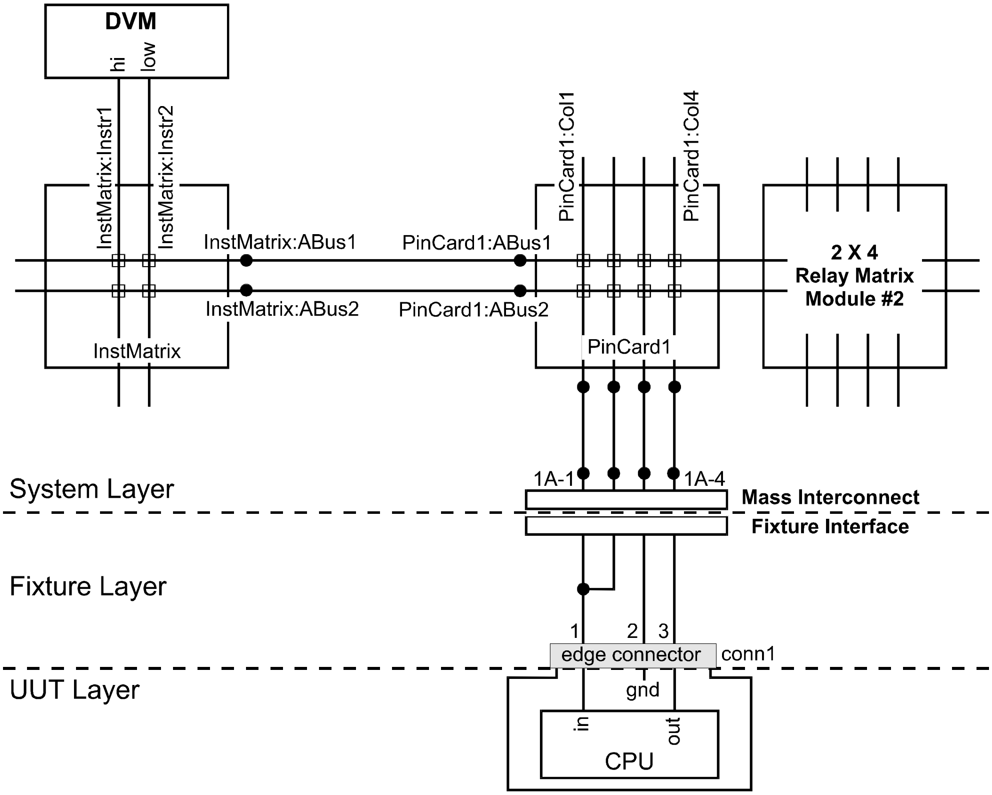
Select the type of topology layer you wish to create.

● UUT Layer
○ Fixture Layer
○ System Layer

Shown below is an example we will work through. Let us begin at a conceptual level and identify the task at hand. Suppose our goal is to connect an instrument to pins on the UUT so the instrument can make a measurement. To provide flexibility in connecting the instrument, an Instrument Matrix module connects the instrument to an analog bus structure connected to two Relay Matrix modules. One of these modules—the one in which you are interested—is connected to a mass interconnect, which is the nexus for connections between the test system and the UUT. From there, fixturing or cabling connects the mass interconnect to the UUT.


Instrument
2 X 2 Instrument Matrix Module
analog bus
2 X 4 Relay Matrix Module #1
2 X 4 Relay Matrix Module #2
desired connection
Mass Interconnect
fixturing/cabling
UUT

The conceptual diagram above lacks details needed to describe real hardware, such as pin numbers and connectors. These details are shown next.



Matching Physical Hardware to Logical Names

Where Do the Names of Switching Paths Come From?

One question upon examining the example above might be, “Where do the names of signal paths used in switching, such as `InstMatrix:ABus1`, come from?” The names of switching paths inside a module are assigned by whoever develops the hardware handler for the module. The Switching Topology Editor lets you use these names to define your test system's topology. The names of other items, such as the pins on connectors, are defined by you and usually reflect the physical characteristics of the item. For example, `conn1-1` is pin 1 of the connector named `conn1`.

In the example above, the hardware handler’s developer chose `InstMatrix` as the name for the 2 X 2 Instrument Matrix Module. In a similar fashion, the first 2 X 4 Relay Matrix Module was named `PinCard1`. Both of these modules contain switching elements that connect rows with columns when they close. The columns in the `InstMatrix` module connect to an instrument, so they are named `InstMatrix:Instr1` and `InstMatrix:Instr2`, and the rows in `InstMatrix` connect to the analog bus, so they are named `InstMatrix:ABus1` and `InstMatrix:ABus2`.

The important thing to realize here is that the intersection of any two of these identifiers is a switching element that can be controlled by the Test Executive during a test. For example, at the intersection of `InstMatrix:Instr1` and `InstMatrix:ABus2` is a relay that, when closed, connects the `hi` side of the DVM to the second analog bus. If this connection is needed during a test, then you could use the Switching Path Editor to tell the Test Executive when to close (and reopen) it.

Switching elements inside module `PinCard1` connect the analog busses with wiring to the mass interconnect, which is the interface between the test system and the cabling/fixturing that connects to test system to the UUT. Connections on the analog bus are denoted the same as their counterparts in the `InstMatrix` module, while columns in `PinCard1` are identified in a more generic sense as `PinCard1:Col1` through `PinCard1:Col4`.

Using Aliases to Simplify the Names of Switching Paths

Although this approach accurately describes the hardware, it lacks convenience for test developers who must remember which connection is which when using the Switching Path Editor. For example, the name `InstMatrix:Instr2` provides no clue as to what that signal path actually is.

The remedy for this is to use aliases. Aliases let you simplify the definition of the hardware. For example, instead of referring to `InstMatrix:Instr2` you could assign it an alias of `DVM_low`. From then on, you could think in terms of “Connect `DVM_low` to. . ." instead of "Connect `InstMatrix:Instr2` (whatever that is!) to. . ."

When Should I Specify Wires?

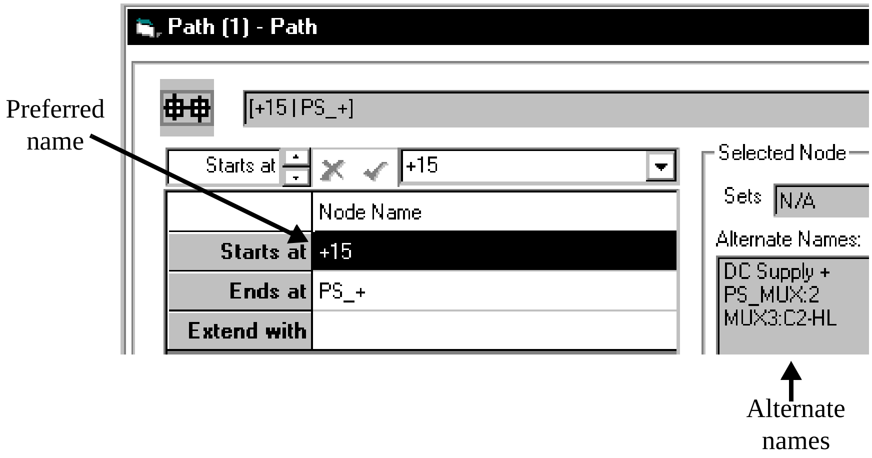
Remember that the Switching Topology Editor also lets you define wires in each layer. An example of a wire is the wire that connects `InstMatrix:ABus1` to `PinCard1:ABus1`. Because this is a connection between modules whose characteristics are modeled in a hardware handler, you should describe it as part of the topology.

What about the wires that connect the DVM to `InstMatrix`? Should they be defined too? Probably not, because defining them offers no additional functionality. Because instruments (and connectors) are not modeled—i.e., they are not defined in hardware handler software—HP TestExec SL is unaware of their characteristics and cannot control them.

What Happens If a Node Has Multiple Names?

Each named electrical point in the switching topology is called a “node.” As described above, the use of aliases and wires lets a node in the topology potentially have more than one name. But if a node has more than one name, which name appears as the “preferred name”—i.e., the name used to construct a switching path—when you use the Path Editor?

An example of this is shown below. Here, a node has several possible names: `+15`, `DC Supply +`, `PS_MUX:2`, and `MUX3:C2-HL`. The preferred name, `+15`, probably is the most meaningful because it describes a major path rather than an individual node somewhere along the path.



Path (1) - Path

[+15 | PS_+]

Starts at: +15

	Node Name
Starts at	+15
Ends at	PS_+
Extend with	

Selected Node

Sets: N/A

Alternate Names:
DC Supply +
PS_MUX:2
MUX3:C2-HL

Preferred name

Alternate names

Part of the value you can add when defining topology is to ensure the “best” name (the name that makes the most sense for your circumstances) for each feature in the topology will appear as the preferred name seen by test developers when they define switching paths.

How Do I Specify the Preferred Name for a Node?

You can specify the preferred name for a node by defining the topology in accordance with the rules the Switching Path Editor uses when it displays the preferred node name. In order of precedence, you should:

Do this . . .	Because . . .
When a node is referenced in more than one topology layer, use the preferred name in the layer that has precedence.	The order for choosing preferred names is UUT layer before fixture layer before system layer.
When a node is associated with a series of aliases—i.e., one alias is aliased to another alias—in the same topology layer, give the preferred name to the first alias in the series.	Within a single layer of topology, the preferred alias in a series of aliases is the first in the series. For example, if `alias1` is aliased to `alias2` that is aliased to `alias3`, the preferred name is `alias1`.
When a node is associated with both a wire and an alias in the same topology layer, give the preferred name to the alias.	Within a single layer of the topology, an alias associated with a node is preferred over a wire associated with the same node.
When a node is associated with multiple aliases (but no wires) in the same topology layer, do whatever you like.	Within a single layer of the topology when multiple aliases exist, the alias chosen will be the last one entered when defining the topology. Because this method tends to be unpredictable, you should not rely upon it.

Defining the System Layer

Continuing with the example begun earlier, you could use the Switching Topology Editor to define the following for the system layer of topology:

Modules:

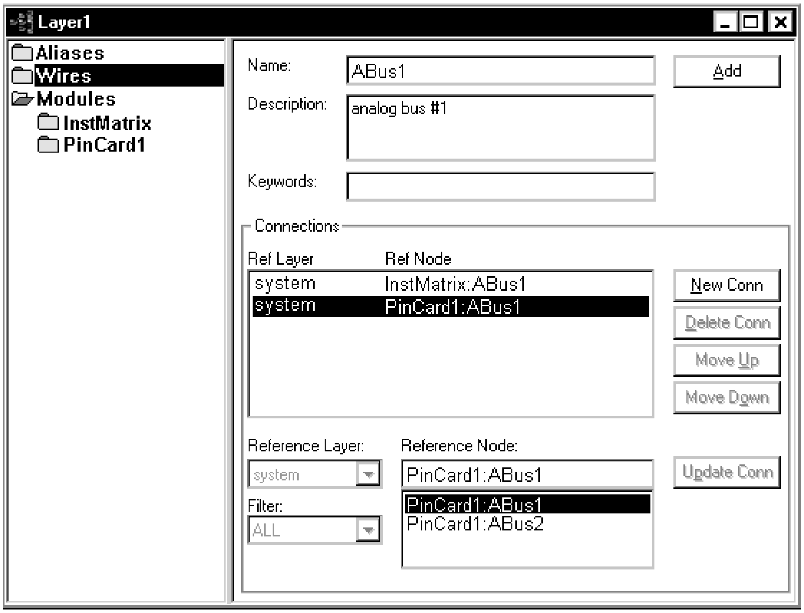
InstMatrix
PinCard1

Wires:

"ABus1" connects `InstMatrix:ABus1` to `PinCard1:ABus1`
"ABus2" connects `InstMatrix:ABus2` to `PinCard1:ABus2`

These wires are necessary because they interconnect switching modules whose topology is known to HP TestExec SL. The topology is known because each module's characteristics are declared in its corresponding hardware handler software (described later). Because each module's topology has been modeled for HP TestExec SL, the Switching Path Editor can control switching elements in it via switching actions in tests.

Using the Switching Topology Editor to specify topology, the definition of the first wire shown above might look like this:



Name: ABus1

Description: analog bus #1

Keywords:

Connections

Ref Layer	Ref Node
system	InstMatrix:ABus1
system	PinCard1:ABus1

Reference Layer: system

Filter: ALL

Reference Node: PinCard1:ABus1

PinCard1:ABus1
PinCard1:ABus2

Aliases:

InstMatrix:Instr1 in the system layer aliased as DVM_hi in the system layer
InstMatrix:Instr2 in the system layer aliased as DVM_lo in the system layer
PinCard1:Col1 in the system layer aliased as 1A-1 in the system layer
PinCard1:Col2 in the system layer aliased as 1A-2 in the system layer
PinCard1:Col3 in the system layer aliased as 1A-3 in the system layer
PinCard1:Col4 in the system layer aliased as 1A-4 in the system layer

Aliases were used here instead of wires because there are no switchable connections. For example, the existence of the cable that connects InstMatrix with the instrument is a given, as is the wiring that connects the columns of relays on PinCard1 with the mass interconnect. If there is no switchable connection to control, it is simplest to use an alias to specify various points along the path.

The benefit of all this work becomes more apparent when you consider how these definitions can simplify the way you specify connections with them. Suppose you want to make a connection between the high terminal on the DVM and a pin on the mass interconnect. Given the definitions of wires and aliases shown above, it could be done as simply as this:

[DVM_hi ABus1 1A-2]

Note

This convention of enclosing the path in brackets and having adjacent nodes separated by spaces is the default used in the Switching Path Editor. To avoid confusion when using the Switching Path Editor, we recommend that you do not use spaces or brackets ([]) when naming features in the topology.

An optional Node Separator entry in the [Switching] section of HP TestExec SL’s initialization file, “<*HP TestExec SL home*\bin\tstexcsl.ini”, lets you specify which character appears as the separator between adjacent nodes for a given installation of HP TestExec SL. For example, Node Separator = | defines a vertical bar as the separator. The separator is not saved with testplans, so if you move a testplan from one test system to another the separator may change.

This describes a connection from one terminal on the DVM, through the relay at the intersection of `InstMatrix:Instr1` and `InstMatrix:ABus1`, across the `ABus` connecting `InstMatrix:ABus1` and `PinCard1:ABus1`, through the relay at the intersection of `PinCard1:ABus1` and `PinCard1:Col2`, and through the wire that connects `PinCard1:Col2` to pin `1A-2` on the mass interconnect. Notice how much more complicated the actual path is than the notation needed to describe it using wires and aliases.

Defining the Fixture Layer

The fixture layer for the previous example might look like this:

Wires:

`conn1-1` in the fixture layer connected to `1A-1` in the system layer
`conn1-1` in the fixture layer connected to `1A-2` in the system layer
`conn1-2` in the fixture layer connected to `1A-3` in the system layer
`conn1-3` in the fixture layer connected to `1A-4` in the system layer

At first glance, you may wonder why these are not defined as aliases. After all, there are no switchable paths in the fixture layer. Notice, however, that both pins `1A-1` and `1A-2` of the mass interconnect are connected to pin 1 of `conn1`. This means that two distinct paths exist to `conn1-1`, depending upon which relay is closed on `PinCard1`. Thus, these should be defined as individual wires and not simply aliases for the same point.

If desired, you also could use a combination of wires and aliases, like this:

Wires:

`conn1-1` in the fixture layer connected to `1A-1` in the system layer
`conn1-1` in the fixture layer connected to `1A-2` in the system layer

Aliases:

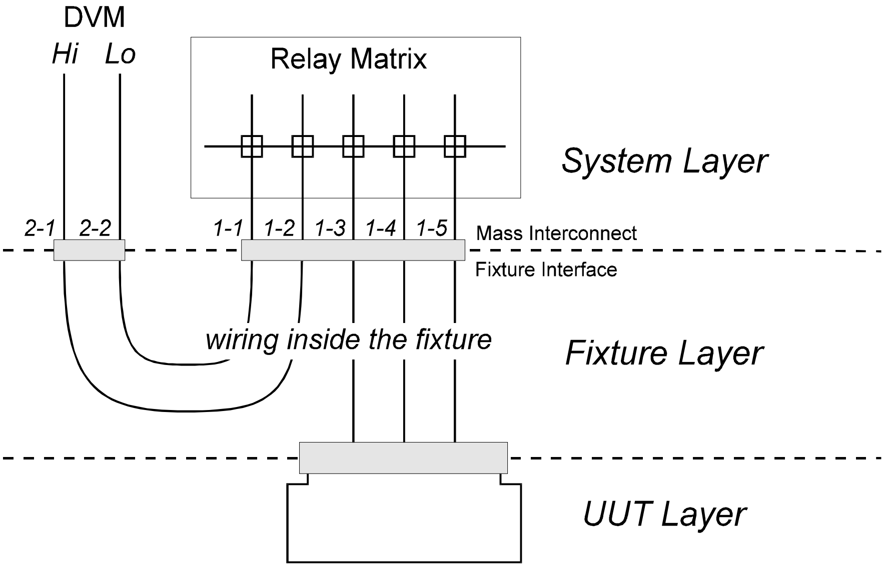
`conn1-2` in the fixture layer aliased as `1A-3` in the system layer
`conn1-3` in the fixture layer aliased as `1A-4` in the system layer

This layer has no modules defined for it because there are no switching modules in the fixture layer for this example. If your fixturing included some form of electronics that was controlled via a hardware handler, you could define it as a module in this layer.

Note

We recommend that all references from the fixture layer to the system layer specify pin identifiers on the mass interconnect and not specify aliases or nodes other than pins on the mass interconnect in the system layer. Following this suggestion lets you alter wiring in the system layer without affecting the fixture.

Shown below is a useful variation on defining topology in the fixture layer (but which will not be a part of the ongoing example). Suppose that instead of using a relay matrix module to connect an external instrument, you connect it to the test system via wiring in the fixture. In other words, when you install the fixture used to test a specific UUT, that fixture contains a connector to which the instrument is attached. The idea here is that by connecting the external instrument to the test system through the mass interconnect, you make the instrument accessible to any relay matrix cards in the test system.



How would you define this topology? Because the connection between the external instrument—”DVM”—and the test system does not contain a switchable path, you could specify the topology as:

Wires:

DVM_hi in the fixture layer (no connections to other layers)
DVM_lo in the fixture layer (no connections to other layers)

Aliases:

DVM_hi in the fixture layer aliased as 1-2 in the system layer
DVM_lo in the fixture layer aliased as 1-1 in the system layer

Defining wires without connections and aliasing them to pins on the mass interconnect makes them equivalent. Thus, a reference to DVM_lo actually means pin 1-1 on the mass interconnect in the system layer.

Defining the UUT Layer

Continuing the example, the topology definition for the UUT layer might look like this:

Aliases:

conn1-1 in the fixture layer aliased as CPU_in in the UUT layer
conn1-2 in the fixture layer aliased as CPU_gnd in the UUT layer
conn1-3 in the fixture layer aliased as CPU_out in the UUT layer

These are all aliases because there is no switchable path; i.e., the aliasing is being done simply for the convenience of specifying CPU_in when using the Switching Path Editor instead of trying to remember what is connected to which pin on conn1 or to pins on the UUT.

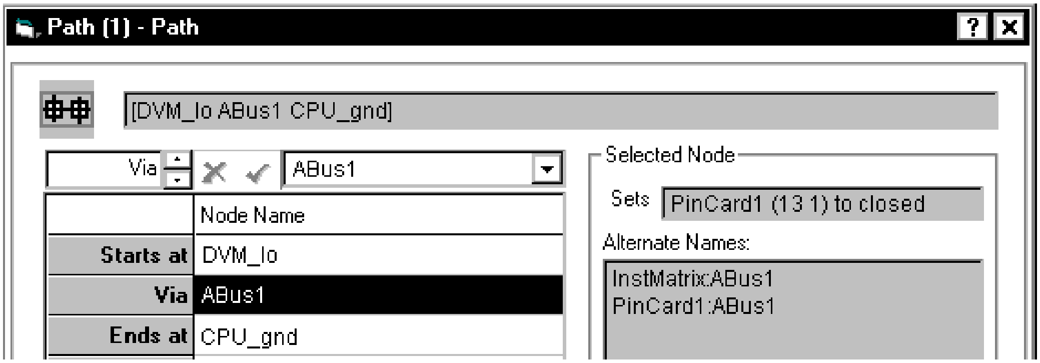
Notice how the aliases are used to alias items in one topology layer with items in another layer. This was necessary because connector *conn1* is the physical interface between the system and UUT layers.

This layer has no modules defined for it because there are no switching modules in the UUT layer for this example. It has no wires defined for it because there are no adjacent nodes—i.e., nodes with a switching element between them—between the fixture and the UUT.

Given the topology defined in all three layers, when creating a test you could use the Path Editor to define a connection between the low terminal on the DVM and ground on the CPU as:

[DVM_lo ABus1 CPU_gnd]

An example of this is shown below.



[DVM_lo ABus1 CPU_gnd]

Via ABus1

	Node Name
Starts at	DVM_lo
Via	ABus1
Ends at	CPU_gnd

Selected Node

Sets: PinCard1 (13 1) to closed

Alternate Names:
InstMatrix:ABus1
PinCard1:ABus1

Using the Switching Topology Editor

To Create a Topology Layer

Use the Switching Topology Editor’s graphical tools to create a topology layer.

1. Click 🗋 in the toolbar or choose File | New in the menu bar.

2. Choose “Topology Layer”.

3. Choose the OK button.

4. Use the Switching Topology Editor Options box to specify which type of topology layer to create.

5. Choose the OK button.

6. When the Topology Layer window appears, use it to define the topology for the layer.

7. (*optional*) If you wish to include summary information about the topology layer, do the following:

   a. Choose File | Revision Information in the menu bar.

   b. Use the Topology Information box to enter summary information in the appropriate fields.

      Tip: For Current Revision, use the Major number to denote large changes to the topology layer, such as adding a number of aliases, wires or modules. Use the Minor number to denote small changes, such as defect fixes or minor enhancements.

   c. Choose the OK button to close the dialog box.

8. Choose File | Save As in the menu bar.

9. Specify a name for the file in which the layer is saved.

   Tip: The names of files for topology layers must have a “.ust” extension—e.g., “system.ust”.

10. Choose the OK button to save the file.

Using Aliases

To Add an Alias

Do the following in the Topology Layer window:

1. Click the Aliases folder in the list area (left pane).

2. Use the editor (right pane) to specify the information for the alias.

The information you must specify for an alias includes:

Name The name of the alias.

Note: We recommend that you do not use spaces or brackets ([]) in names because that makes switching paths more difficult to read.

Description A description of the alias.

Keywords One or more keywords, separated by commas, that aid users when searching for this alias among all the possible aliases. Keywords are used by the Filter feature.

Reference Node An existing name that specifies a node in a topology layer.

Note: Click the arrow to the right of Filter to invoke a list of keywords that restrict the search criteria in the Reference Node list.

Reference Layer The topology layer that contains the reference node.

3. Choose the Add button.

To Modify an Alias

1. In the Topology Layer window, click to open the Aliases folder in the list area (left pane) if it is not already open.

2. Click the alias you wish to modify.

   Tip: You can use the list of keywords under Filter to reduce the length of the list of reference nodes that appears.

3. Use the editor (right pane) to modify the information for the existing alias.

4. Choose the Update button.

To Delete an Alias

1. In the Topology Layer window, click to open the Aliases folder in the list area (left pane) if it is not already open.

2. Click the alias you wish to delete.

3. Do either of the following:

   - Press the Del key.

     - *or* -

   - Choose Edit | Delete in the menu bar.

4. Choose the Update button.

Using Wires

To Add a Wire

Do the following in the Topology Layer window:

1. Click the Wires folder in the list area (left pane).

2. Use the editor (right pane) to specify the information for the wire.

   The information you must specify for a wire includes:

   Name The name of the wire.

   We recommend that you do not use spaces or brackets ([]) in names because that makes switching paths more difficult to read.

   Description A description of the wire.

   Keywords One or more keywords, separated by commas, that aid users when searching for this wire among all the possible wires. Keywords are used by the Filter feature.

Connections One or more nodes to which the wire is electrically connected.You can click:

New Conn—add a new connection to the list.

Delete Conn—remove the selected connection from the list.

Move Up—promote the position of the selected connection in the list.

Move Down—demote the position of the selected connection in the list.

Reference Node An existing name that specifies a node in a topology layer. Note: Click the arrow to the right of Filter to invoke a list of keywords that restrict the search criteria in the Reference Node list.

Reference Layer The topology layer that contains the reference node.

3. Choose the Add button.

To Modify a Wire

1. In the Topology Layer window, click to open the Wires folder in the list area (left pane) if it is not already open.
2. Click the wire you wish to modify.

   Tip: You can use the list of keywords under Filter to reduce the length of the list of reference nodes that appears.
3. Use the editor (right pane) to modify the information for the existing wire.
4. Choose the Update button.

To Delete a Wire

1. In the Topology Layer window, click to open the Wires folder in the list area (left pane) if it is not already open.

2. Click the wire you wish to delete.

3. Do either of the following:

   - Press the Del key.

     - *or* -

   - Choose Edit | Delete in the menu bar.

4. Choose the Update button.

Using Modules

To Add a Module

Do the following in the Topology Layer window:

1. Click the Modules folder in the list area (left pane).

2. In the editor (right pane), click the Library field and either type the name of the library file that contains the module's instrument driver/handler or use the Browse button to find the correct file.

3. If you are using a VXI*plug&play* driver, enter the Prefix (described below) that identifies the instrument.

4. Choose the Add button to load the parameter block for the instrument.

5. Do the following for each parameter in the list under Parameter Block:

   a. Select the parameter.

   b. Choose the Edit button.

   c. Specify the information to be passed in the parameter.

6. Enter the remaining information for the module (described below).

7. Choose the Update button.

The information you must specify for a module includes:

Name A unique name for the module.

We recommend that you do not use spaces or brackets ([]) in names because that makes switching paths more difficult to read.

Disable Enable this box to have the Test Executive ignore the module, such as when you remove it for calibration.

Description A description of the module.

Prefix An identifier that is generally used with VXI*plug&play* instruments to identify the type of instrument. Enter the name of the instrument as it appears in calls to the VXI*plug&play* driver; e.g., calls to HP663x2-series instruments begin with “hp663x2” (as in “hp663x2_init”) so that is what you should enter.

Library The name of the library (DLL) that contains the hardware handler for the module. In the case of a VXI*plug&play* instrument, specify the name of the DLL in which the VXI*plug&play* driver for the instrument resides.

Parameter block A list of parameters passed to the module in its parameter block. If the DLL for the module is not found, the list will be empty.

To Modify a Module

1. In the Topology Layer window, click to open the Modules folder in the list area (left pane) if it is not already open.

2. Click the module you wish to modify.

3. Use the editor (right pane) to modify the information for the existing module.

4. Choose the Update button.

To Delete a Module

1. In the Topology Layer window, click to open the Modules folder in the list area (left pane) if it is not already open.

2. Click the module you wish to delete.

3. Do either of the following:

   - Press the Del key.

     - *or* -

   - Choose Edit | Delete in the menu bar.

4. Choose the Update button.

Duplicating an Alias, Wire, or Module

Instead of specifying the characteristics of similar aliases, wires, or modules multiple times, you can copy an existing item and then rename it or modify its characteristics.

1. With a switching topology layer loaded, select an existing alias, wire, or module in the left pane (list area) of the Switching Topology Editor.

2. Choose Edit | Duplicate in HP TestExec SL's menu bar.

The duplicate entry will appear below the existing entry.

5

Working with Libraries, Datalogging, Symbol Tables, & Auditing

This chapter describes how to use libraries of actions and tests to promote code reusability, datalogging to collect data during testing, symbol tables to store global variables, and auditing features to track software revisions.

For related overview topics, see Chapter 3 in the *Getting Started* book.

Using Test & Action Libraries

For an overview of test and action libraries, see “About Test & Action Libraries” in Chapter 3 of the *Getting Started* book.

How Keywords Simplify Finding Items in Libraries

When you save an action definition or a test definition in a library, you have the option of specifying one or more “keywords” with the definition. A keyword is 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 the action does or how it is used. Once you have searched for items whose keywords seem appropriate, you can inspect the resultant list and choose the correct item from it. Or, you can set up a new search and try again.

Because the number of actions you create can grow quite large, when working with actions (as opposed to tests) you can use an additional feature called “master keywords.” Master keywords are keywords stored in an editable predefined list, which lets you quickly choose a keyword when creating actions. A major benefit of master keywords is that you can standardize the list for consistency when finding actions in libraries.

Having meaningful keywords assigned to items in libraries lets you use HP TestExec SL’s browsing tools to find items quickly. Although specifying keywords requires slightly more effort initially, over time you will benefit from enhanced code reuse.

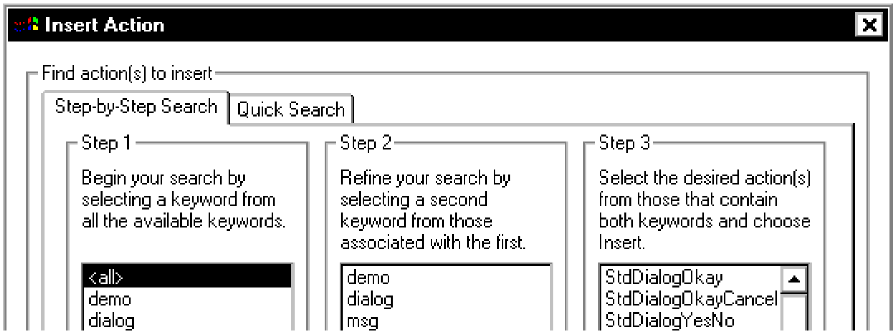
Searching for Items in Libraries

Note

Before you can search libraries, you must set up their search paths, as described under “Specifying the Search Path for Libraries.”

Searching for Actions in a Library

HP TestExec SL provides two ways to search for actions to insert into tests. The first, which is on the Step-byStep Search tab and shown below, uses multiple keywords and provides step-by-step guidance, which makes it useful for beginners.



Insert Action

Find action(s) to insert

Step-by-Step Search | Quick Search

Step 1

Begin your search by selecting a keyword from all the available keywords.

<all>
demo
dialog

Step 2

Refine your search by selecting a second keyword from those associated with the first.

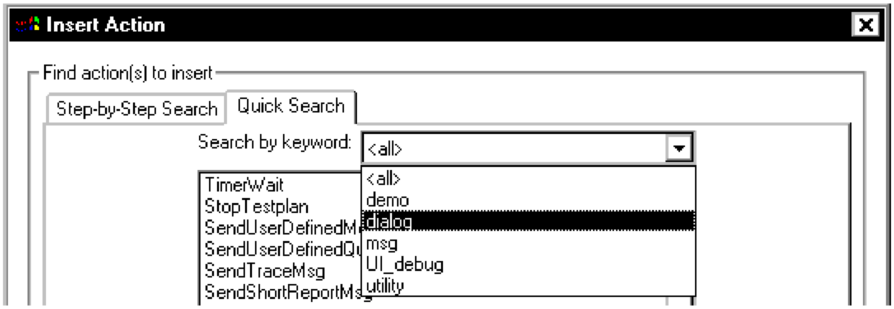
demo
dialog
msg

Step 3

Select the desired action(s) from those that contain both keywords and choose Insert.

StdDialogOkay
StdDialogOkayCancel
StdDialogYesNo

The second search method, which is on the Quick Search tab and shown next, uses a single keyword and can be faster for experienced users.



Insert Action

Find action(s) to insert

Step-by-Step Search | Quick Search

Search by keyword: <all>

<all>
demo
dialog
msg
UI_debug
utility

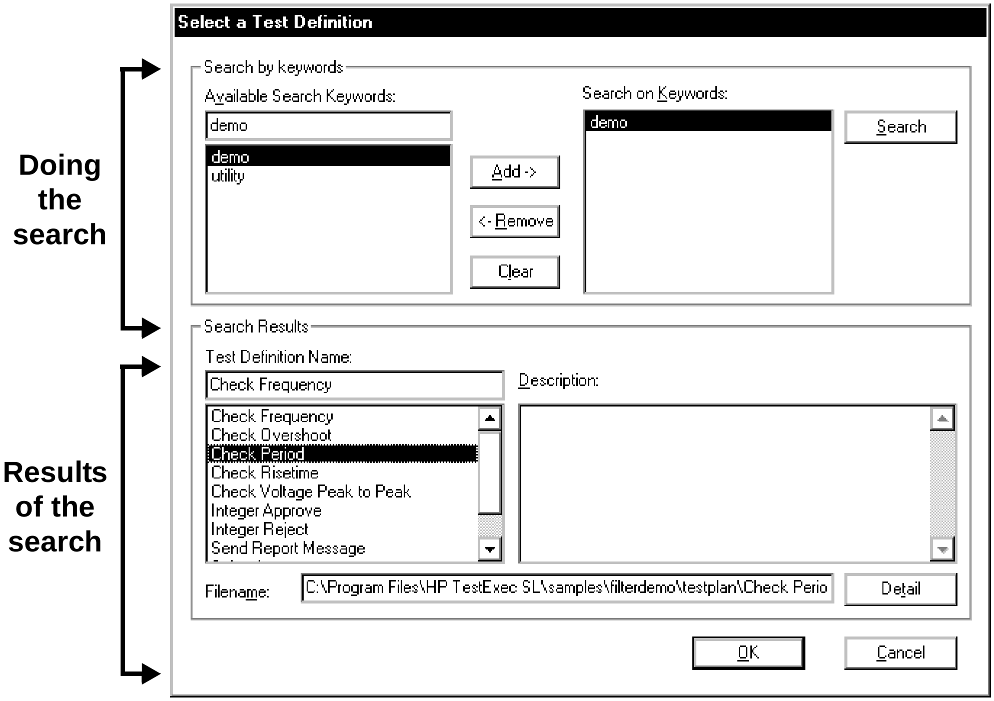
TimerWait
StopTestplan
SendUserDefinedM
SendUserDefinedQ
SendTraceMsg
SendShortReportMs

Note

The name of the action searched for is the name the action was given when it was created with the Action Definition Editor, which is not necessarily the same as the name of the file in which the action definition resides.

Searching for Tests in a Library

A common task when creating testplans is to search for tests to use in them. The search mechanism used to search for a saved test definition so you can insert it into a testplan is shown below.



Select a Test Definition

Search by keywords

Available Search Keywords: demo

demo
utility

Search on Keywords:

demo

Search Results

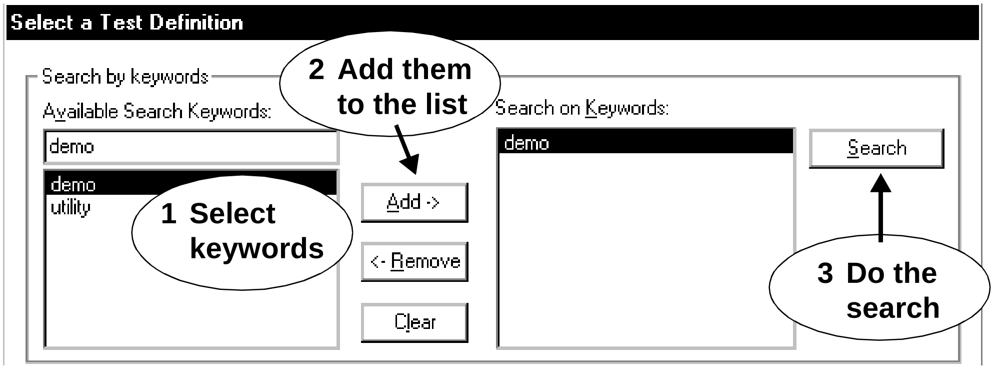
Test Definition Name: Check Frequency

Check Frequency
Check Overshoot
Check Period
Check Risetime
Check Voltage Peak to Peak
Integer Approve
Integer Reject
Send Report Message

Description:

Filename: C:\Program Files\HP TestExec SL\samples\filterdemo\testplan\Check Perio

The general procedure for doing a search for a test is shown below.



Select a Test Definition

Search by keywords

Available Search Keywords: demo

demo
utility

Search on Keywords:

demo

If desired, you also can:

- Choose the Remove button to remove a selected keyword from the search list.
- Choose the Clear button to remove all keywords from the search list.

Strategies for Searching Libraries

Built into the Test Executive's graphical tools are several features that help you search the contents of libraries for a specific routine. Your general strategy when searching should be to reduce the list of matches as quickly as possible, until only a few potential items of interest must be browsed to find the desired one.

To quickly find the item of interest, you can:

- Limit the list of library directories of each type to be searched.

  By restricting the list of directories to those most likely to contain entries useful for the test under development, you can eliminate many unnecessary entries before beginning the search.

- Use keywords to narrow the search.

  Select one or more keywords from those known to be in the entries, and only library entries with all the selected keywords will be displayed in the list of matches.

- Type the first few characters of the name of the desired entry to position the list of entries to the appropriate part of the alphabetized list.

The features used to search libraries work best when the libraries are carefully defined and organized. Where possible, do the following:

- Organize library directories such that the entries in them are logically related and likely to be needed in similar testing situations.
- Be sure that the names of libraries and the entries in them reflect their contents.

- Use meaningful keywords when describing the entries in libraries.
- Provide related entries with similar prefixes on their names (which improves HP TestExec SL's ability to sort by name).
- Use HP TestExec SL's Action Libraries box or Test Libraries box to find whichever kind of routine you need.

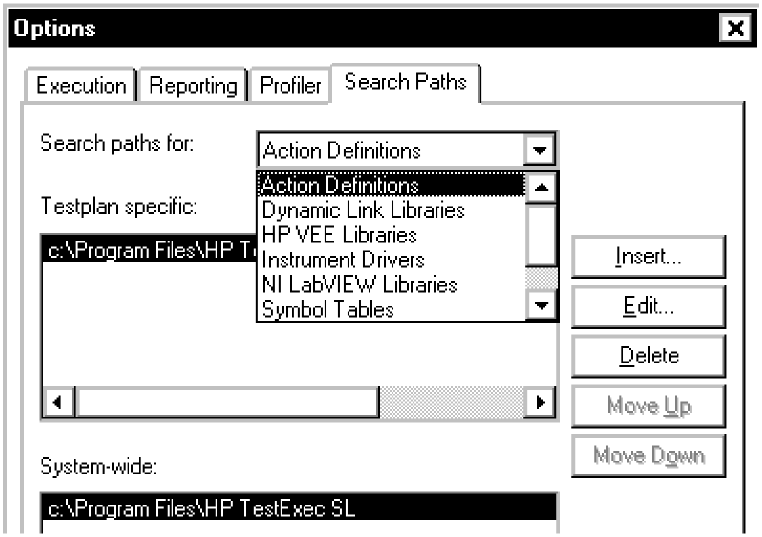
Specifying the Search Path for Libraries

HP TestExec SL lets you specify the search paths for action definitions, dynamic link libraries (DLLs), HP VEE libraries, instrument drivers, National Instruments LabVIEW libraries, symbol tables, test definitions, and layers in the switching topology. You can specify these paths at two levels:

Testplan-specific Paths that are specific to whichever testplan currently is loaded, and override paths specified at the System-wide level.

System-wide Default paths that apply unless they are overridden at the Testplan-specific level. If you create a new testplan and do not specify specific paths for it, these defaults will be used.

In either case, you use a dialog box similar to the one shown below.



Search paths for: Action Definitions

Action Definitions
Dynamic Link Libraries
HP VEE Libraries
Instrument Drivers
NI LabVIEW Libraries
Symbol Tables

Testplan specific:

c:\Program Files\HP T

System-wide:

c:\Program Files\HP TestExec SL

Note

Search paths are searched in the order shown in the lists under Testplan-specific and System-wide. This means that if your testplan uses a specific DLL, and multiple instances of that DLL exist on your test system, only the first instance of the DLL to be found will be used.

Given the above, modifying the order in which the paths are searched potentially influences which items are found. You can use the Move Up and Move Down buttons or “drag and drop” with the mouse to reorder the search paths in the lists.

To Specify System-Wide Search Paths for Libraries

1. *With no testplan loaded*, choose Options | System Options in the menu bar.

2. In the list to the right of `Search paths for:`, choose which kind of system-wide search path you wish to specify; i.e., a search path for action definitions, dynamic link libraries, etc.

3. Choose the Insert button.

4. When the Insert Path box appears, either type a search path directly into the data entry field or choose the Browse button and use the graphical browser to specify a search path.

5. Choose the OK button to save the new path in the list under System-wide.

To Specify Testplan-Specific Search Paths for Libraries

1. *With a testplan loaded*, click ⓘ in the toolbar or choose View | Testplan Options in the menu bar.

2. In the Options box, choose the Search Paths tab.

3. Click to select an insertion point in the list of search paths under Testplan-specific.

Note

If you click in the list of System-wide search paths, you also can specify those here. Be aware, though, that changes made here become the new system defaults. Use whichever method you prefer.

4. Choose the Insert button.

5. When the Insert Path box appears, either type a search path directly into the data entry field or choose the Browse button and use the graphical browser to specify a search path.

6. Choose the OK button to save the new path in the list under Testplan-specific.

To Remove a Path from the List of Search Paths

1. Click ⓘ in the toolbar or choose View | Testplan Options in the menu bar.

2. In the Options box, choose the Search Paths tab.

3. In either of the lists of search paths, click the search path to be deleted.

4. Choose the Delete button.

5. Choose the OK button.

Using Search Paths to Improve Testplan Portability

Having two levels of search paths, testplan-specific and system-wide as described above, is especially useful when testplans must be transportable across test systems. For example, if you specify only system-wide search paths, a testplan moved from one system to another will automatically use the default search paths for the new system. On the other hand, specifying testplan-specific search paths lets you override the defaults as needed, so you know exactly which files a given testplan will use.

Using Datalogging

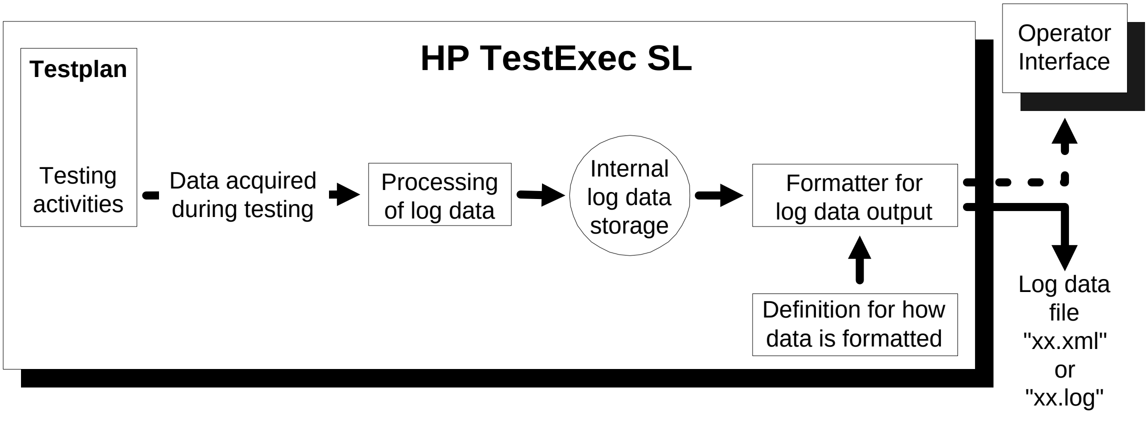
This section discusses datalogging options, disabling datalogging for individual tests, using datalogging with Q-STATS programs, datalogging files and their formats, and how to change between datalogging formats.

What Happens During Datalogging?

Datalogging automatically collects data about tests *that have pass/fail limits specified for them* when a testplan runs. Subsequent study of this data can help you improve the testing and manufacturing process and track the testing done on a particular UUT.

The system writes a new file of datalogging information each time a testplan or a loop in a testplan runs. The system automatically names each file with a unique hexadecimal name derived from the system date and time plus an extension of “.xml” or “.log” depending upon the format chosen for it.

The flow of data is shown below. First, data acquired during testing is formatted using a definition for internal data, and stored internally. Next, the internally stored log data is reformatted using a definition for output data, and saved in an external data file for subsequent analysis.


HP TestExec SL
Testplan
Testing activities
Data acquired during testing
Processing of log data
Internal log data storage
Formatter for log data output
Definition for how data is formatted
Operator Interface
Log data file "xx.xml" or "xx.log"

The diagram above also shows an optional path by which log data can be sent to an operator interface for post-processing. To simplify this, HP TestExec SL provides an ActiveX™ datalogging control for use in operator interfaces written in Visual Basic; see Chapter 3 in the *Customizing HP TestExec SL* book.

What is the Behavior & Format for Logged Data?

Datalogging can have the following behaviors:

TxSL Log records conform to a proprietary log record schema used by HP TestExec SL. This is the default behavior and the one recommended for general use.

HP 3070 Log records conform to a subset of the records produced by a HP 3070-family board test system. This means you can use Derby Associates Q-STATS II or HP Pushbutton Q-STATS to do statistical analysis of log data.

The choice of behavior determines the names of the log records but not the format of the data in the fields inside them.

Note If desired, you can modify the records and fields that appear in datalogging files. See Chapter 3 in the *Customizing HP TestExec SL* book.

For any given behavior, you have a choice of formats that determine the presentation of the data. These are:

Spreadsheet Datalogging files are written with each record on a separate line and the fields separated by commas. This format, which is readable by most spreadsheet or database programs, lets you develop your own methods for analyzing log data with the functions available in a spreadsheet.

XML Datalogging files are written in XML (eXtensible Markup Language) format. Because XML is a standard for describing data, the use of XML lets you use commercially available tools to view or process datalogging files.

HP 3070 Datalogging files are written in the format used by the HP 3070 family of board test systems.

Controlling How Datalogging Works

To Set the Datalogging Options for an Entire Testplan

1. Click ⓘ in the toolbar or choose View | Testplan Options in the menu bar.

2. When the Testplan Options box appears, choose its Reporting tab.

3. Choose the desired options.

   The global datalogging options for a testplan are:

   Enabled When this box is checked, datalogging is enabled for the current testplan, and datalogging is disabled when the box is unchecked.

   Log Report Information When this box is checked, any messages that appear in the Report window will be included in the datalogging log file.

Log Level Determines how much and which kind of data is collected during datalogging, as follows.

`All`—Logs a header for each testplan that was executed, plus information about how its tests passed or failed.

`None`—Logs a header for each testplan that was executed, but does not identify whether its tests passed or failed.

`Failures`—Logs a header for each testplan that failed, plus information about how its tests failed.

`Sampled`—Logs a header for each testplan that was executed. Logs information for all tests that fail. Also, logs information for a specified percentage of passing tests.

`Sample rate %`—Sets the percentage of passing tests to be sampled when the testplan is run.

Log Directory Specifies the directory that will hold the datalogging files. (By default, log data is stored in directory “\logdir”.) If the system cannot access the indicated directory, log files will be temporarily placed in the directory specified by the system variable TMP. (Usually this is the “\temp” directory.) The system will attempt to move the log files to the current log directory each time a testplan runs.

4. Choose the OK button.

To Change the Datalogging Options for an Individual Test

Options on the Reporting tab of the Testplan Options box (View | Testplan Options) let you specify the global behavior of datalogging across all of the tests in a testplan. As described below, you can also change some options for individual tests in a testplan.

1. Click a test of interest in the left pane of the Testplan Editor window.

2. Choose the Options tab in the right pane of the Testplan Editor window.

3. Choose the desired options for the test.

   The datalogging options for individual tests are:

   Generate unique names for datalogging when looping
   Check this box and the selected test's data will be logged under a unique name each time the test is executed inside a loop (such as `For...Next`).[a]

   Pass/Fail only affects 'On Fail Branch To'
   Check this box to disable datalogging for the selected test.

   Be aware that this option does more than turn off datalogging for an individual test. It also disables any pass/fail messages normally sent to the Report window, cancels any effect of the test on global pass/fail information, and causes statement tracking to be skipped for the test. However, the "On Fail Branch To" feature still works.

   Override the Test Name for Datalogging
   If you want to have the selected test logged under a different name, check this box and specify the new name in the data entry field to the right of `New Test Name for datalogging:`.

   [a] Unique names are generated in log data by appending a unique integer onto the name of the test; e.g., "Test1", "Test2".

4. Choose the OK button.

To Select the Datalogging Behavior and Format

You can switch among the various datalogging behaviors and formats (described earlier under "What is the Behavior and Format for Logged Data?") by editing the HP TestExec SL initialization file, as follows.

1. Use a text editor, such a WordPad in its text mode, to open file "<*HP TestExec SL home*>\bin\tstexcsl.ini".

2. Locate the [Data Log] section in the initialization file and specify which set of behavior and format files to use. (The [Data Log] section has comments that describe how to do this.)

3. Save the modified inialization file and exit the text editor.

For more information about datalogging formats and customizing datalogging, see Chapter 3 in the *Customizing HP TestExec SL* book.

Using Datalogging with a Spreadsheet

Because they are a familiar tool in the business world, spreadsheets provide a useful and straightforward way to examine or manipulate data acquired via datalogging. The following topics describe how to use datalogging files with a spreadsheet.

To Configure Datalogging for Use With a Spreadsheet

Note If you need more information about the log records described below or about using the Datalogging Configuration Editor, see the online help for the Datalogging Configuration Editor.

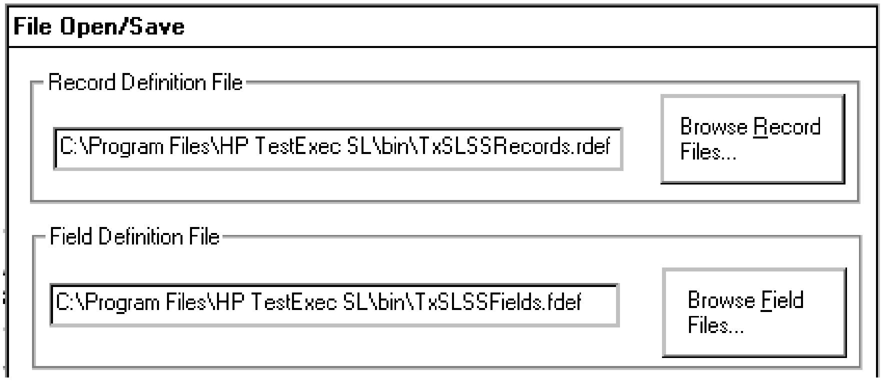
The basic approach to using datalogging with a spreadsheet is to use only two types of records for datalogging information that appears in the spreadsheet[1]. The first is a LogBatch record, only one of which will be produced, that contains a row of titles assigned to columns in the spreadsheet. The second is a LogLimit record that contains information about the test's pass/fail status and its limits. Because *N* LogLimit records are produced—one for each execution of a limit checker—the contents of each LogLimit record populate one line in the spreadsheet.

1. Use the Start menu in Windows to run the Datalogging Configuration Editor.

[1] Other types of records are present, but they contain no fields.

2. Choose File | Open in the Datalogging Configuration Editor’s menu bar.

3. Specify the record and field definition files shown below, which contain predefined definitions you may be able to use as-is.


File Open/Save

Record Definition File: C:\Program Files\HP TestExec SL\bin\TxSLSSRecords.rdef

Field Definition File: C:\Program Files\HP TestExec SL\bin\TxSLSSFields.fdef

4. Choose the Open button.

5. If the fields specified to appear in the LogLimit record do not meet your needs, specify which fields you want in that record. The top-to-bottom order in which they appear in the editor is the order in which they will appear in columns in the spreadsheet.

You now have the option of specifying different headers, trailers, and separators for the data sent to the spreadsheet. The following are true when you use the recommended record and field definition files or whenever you choose the AutoCreate Headers, Trailers and Separators button to restore the headers, trailers, and separators to their default values for spreadsheet compatibility.

- The trailer for the LogLimit record is a line feed, and the separator between items is a comma. This results in fields separated by commas and instances of the LogTest record separated by line feeds, which makes the format compatible with a spreadsheet.

- The header created for the LogBatch record is a comma-separated, quoted, ordered list of field names from the LogTest record. This provides the heading for columns in the spreadsheet.

- All other records headers, trailers and separators are empty strings.

6. Do the following if you wish to modify the headers, trailers, and separators:

   a. Select a record in the left pane of the editor.

   b. Choose the Edit Record Definitions button.

   c. Use the drop-down list adjacent to Record Name to choose the LogBatch or LogLimit records, and then modify their headers, trailers, and separators as desired.

      Suppose you want to make the field separator a tab. Tabs are specified in headers, trailers and separators by `\t`. Type two characters: a backslash followed by “t”. Similarly, you can specify a new line by entering `\n`.

   d. Choose the Done button.

7. Save the results.

   You have two choices when saving modified files. First, you can use File | Save to resave them under their original names, which overwrites the original files[1]. Second, you can use File | Save As to save them under alternate names, which preserves the original files[2].

8. Choose File | Exit to close the Datalogging Configuration Editor.

[1] If you installed HP TestExec SL with the Custom option and chose to install backup copies of the configuration files, you can always find copies of the original files in directory “<*HP TestExec SL home*>\DefaultConfiguration”.

[2] If you change the names of the files, edit the [Data Log] section of file “<*HP TestExec SL home*>\bin\tstexcsl.ini” so it specifies the new files.

What’s Inside a Datalogging File Formatted for Spreadsheets?

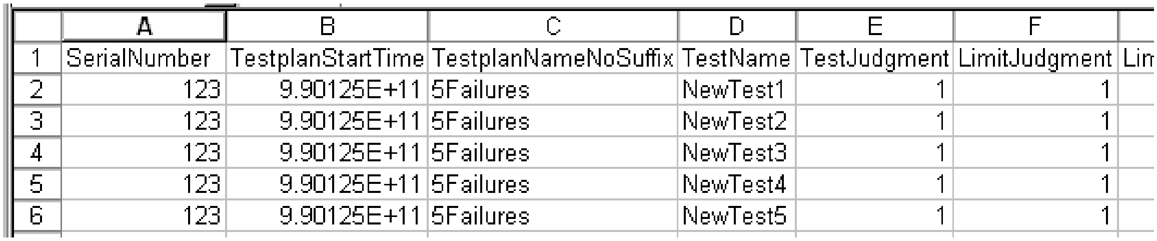
A sample of the datalogging file that HP TestExec SL generates for use with spreadsheets—i.e., when File Type is set to “Spreadsheet”—is shown below[1].

```
"SerialNumber","TestplanStartTime","TestplanNameNoSuffix","TestName",
"TestJudgment","LimitJudgment","LimitLastMeasuredValue","LimitHighLim
it","LimitLowLimit"
123,990125105834,5Failures,NewTest1,1,1,-1,1,1
123,990125105834,5Failures,NewTest2,1,1,-2,2,2
123,990125105834,5Failures,NewTest3,1,1,-3,3,3
123,990125105834,5Failures,NewTest4,1,1,-4,4,4
123,990125105834,5Failures,NewTest5,1,1,-5,5,5
```

The first row contains headings for the columns in which items in subsequent rows appear[2]. Each subsequent row contains the results for a single test. For example, in the second row the value of “TestName” is “NewTest2”, the value of LimitHighLimit is “2”, etc.

How Does the Data Appear in a Spreadsheet?

When loaded into a spreadsheet as comma-delimited data, the rows and columns arrange themselves into a spreadsheet’s representation of data in a grid, as shown in the excerpt below.



	A	B	C	D	E	F	
1	SerialNumber	TestplanStartTime	TestplanNameNoSuffix	TestName	TestJudgment	LimitJudgment	Lin
2	123	9.90125E+11	5Failures	NewTest1	1	1	
3	123	9.90125E+11	5Failures	NewTest2	1	1	
4	123	9.90125E+11	5Failures	NewTest3	1	1	
5	123	9.90125E+11	5Failures	NewTest4	1	1	
6	123	9.90125E+11	5Failures	NewTest5	1	1	

Notice how each column has a heading that identifies its contents. Each row that follows the row of headers contains the results from a single test.

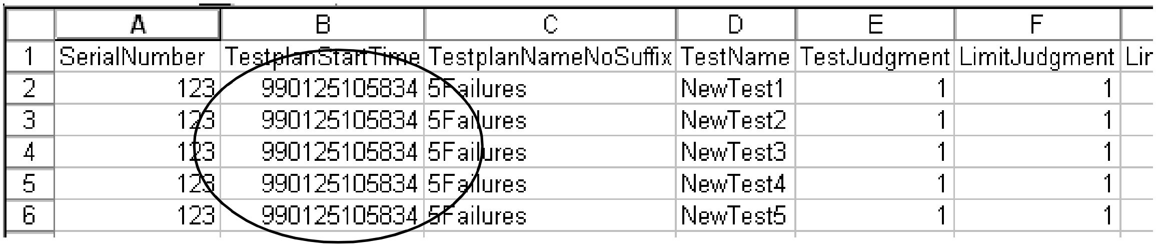
[1] The first three lines are a single row wrapped as-is because the row of data is too long to fit on this page.

[2] The names of columns correspond to names in the datalogging schema described in the online help for the Datalogging Configuration Editor.

Why You May Need to Reformat the Data

Notice how the values of “TestplanStartTime” in the example above are formatted in scientific notation. When importing the data, the spreadsheet attempted to format this value, which is derived from the system date and time. However, the value was formatted incorrectly because it is in a non-standard format that is unknown to the spreadsheet.

Reformatting the cells in which values for “TestplanStartTime” appear returns the data to its original representation in the datalogging file. The example below shows those cells reformatted as numbers without decimal places.



	A	B	C	D	E	F	
1	SerialNumber	TestplanStartTime	TestplanNameNoSuffix	TestName	TestJudgment	LimitJudgment	Lir
2	123	990125105834	5Failures	NewTest1	1	1	
3	123	990125105834	5Failures	NewTest2	1	1	
4	123	990125105834	5Failures	NewTest3	1	1	
5	123	990125105834	5Failures	NewTest4	1	1	
6	123	990125105834	5Failures	NewTest5	1	1	

This example shows how you may need to reformat the data from datalogging files into a format that suits your needs. This is especially true if you customize datalogging to acquire data in formats unique to your testing environment.

For more information about datalogging formats and customizing datalogging, see Chapter 3 in the *Customizing HP TestExec SL* book.

To Import a Datalogging File into Microsoft Excel 97

Microsoft Excel 97’s Text Import Wizard simplifies the task of importing datalogging files that contain comma-delimited data.

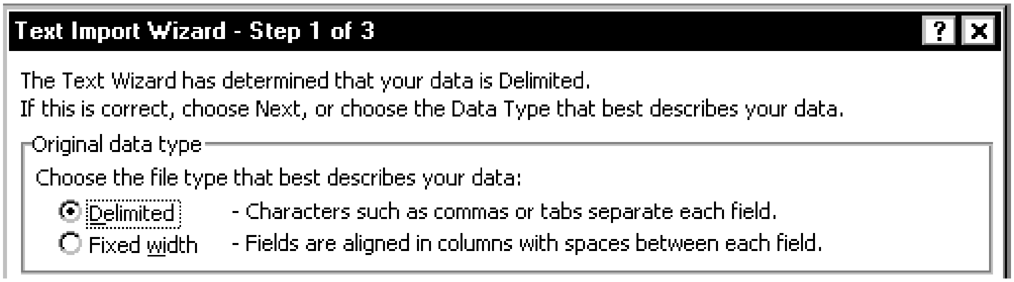
1. Choose File | Open in Excel’s menu bar.

2. Use the Open box to locate and open the datalogging file of interest.

Note

You may need to set `Files of type:` to “All Files (*.*)” to make your file visible.

3. In Step 1 of the Text Import Wizard, verify that the “Delimited” option is chosen, as shown below.



Text Import Wizard - Step 1 of 3

The Text Wizard has determined that your data is Delimited.
If this is correct, choose Next, or choose the Data Type that best describes your data.

Original data type

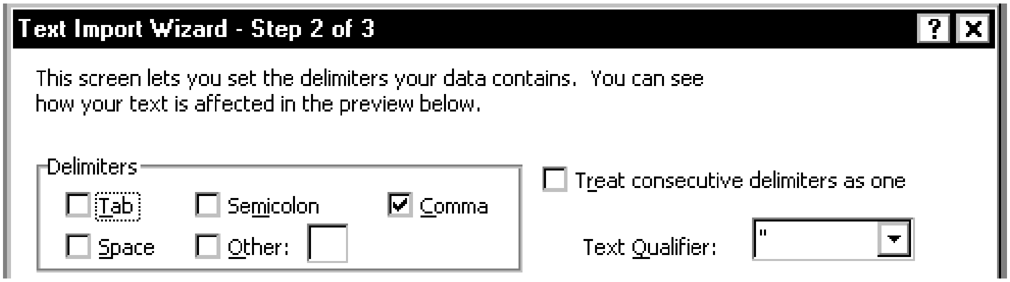
Choose the file type that best describes your data:

● Delimited - Characters such as commas or tabs separate each field.

○ Fixed width - Fields are aligned in columns with spaces between each field.

4. Choose the Next button.

5. In Step 2 of the Text Import Wizard, the delimiter between data items, as shown below. This example assumes you are using comma-delimited data.



Text Import Wizard - Step 2 of 3

This screen lets you set the delimiters your data contains. You can see how your text is affected in the preview below.

Delimiters

□ Tab □ Semicolon ☑ Comma

□ Space □ Other: □

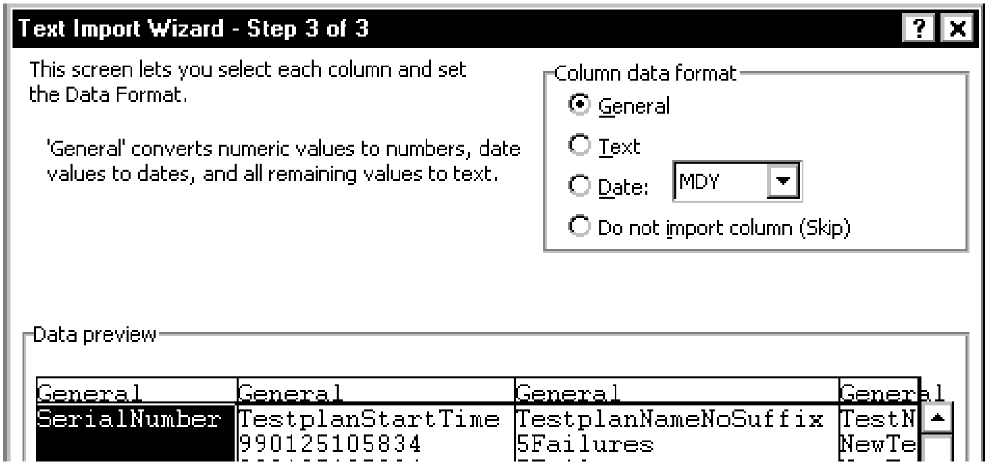
□ Treat consecutive delimiters as one

Text Qualifier: "

6. Choose the Next button.

7. If you have specific formatting needs addressed by Step 3 of the Text Import Wizard, which is shown below, select columns and specify how

they should be formatted. This step also lets you specify columns to be skipped instead of imported.



Text Import Wizard - Step 3 of 3

This screen lets you select each column and set the Data Format.

'General' converts numeric values to numbers, date values to dates, and all remaining values to text.

Column data format

● General
○ Text
○ Date: MDY
○ Do not import column (Skip)

Data preview

General	General	General	General
SerialNumber	TestplanStartTime	TestplanNameNoSuffix	TestN
	990125105834	5Failures	NewTe

8. Choose the Finish button.

Using Datalogging with Q-STATS Programs

When you use HP 3070-style datalogging records, you can use Derby Associates Q-STATS II or HP Pushbutton Q-STATS to do statistical analyses of log data. This section describes how to pass limits to these programs and restrictions on the names of tests.

To Set the Learning Feature & Pass Limits Information

You must pass limits information to the Q-STATS program to construct accurate histograms from the data. To pass limits, you must run the testplan once with the learning feature set to “on.” This setup tells the Q-STATS program that you will pass data limits as well as values.

1. Choose Options | Testplan Options in the menu bar.

2. When the Testplan Options box appears, choose its Reporting tab.

3. Turn on datalogging by enabling the Enabled check box under Datalogging.

4. Set the datalogging level to “all” by enabling the All radio button to the right of Log Level:.

5. Choose the Execution tab in the Testplan Options box.

6. Select Ignore All Failures under Sequencer Halting on the Execution tab.

7. Run the testplan.

Note Any time you change the test limits, you must re-run the testplan with learning set to “on” as described above.

Restrictions on the Names of Tests

Q-STATS II and HP Pushbutton Q-STATS each impose restrictions on the test names that you choose within HP TestExec SL:

- For HP Pushbutton Q-STATS, you must not use slashes (/ or \) in test names.

- For Q-STATS II, only the first 40 characters of the test name are significant.

Managing Datalogging Files

If you set the datalogging level to “all”, HP TestExec SL can quickly fill up the disk that contains the datalogging directory. See “Managing Temporary Files” in Chapter 6 for more information.

Troubleshooting Problems with Datalogging

If datalogging fails to work as expected, consider the following:

- If there is no datalogging information, do you have datalogging enabled? See the description of the `Enabled` option under “To Set the Datalogging Options for an Entire Testplan” earlier in this chapter.

- If there is no datalogging information, does your testplan include tests whose pass/fail limits are checked? Tests must be evaluated against pass/fail criteria to produce datalogging information. See “To Specify the Limits for a Test” in Chapter 2 of the *Using HP TestExec SL* book.

- If datalogging information is present but not what you need, are the options for the log level set correctly; i.e., are you logging the right information? See the description of the `Log Level` option under “To Set the Datalogging Options for an Entire Testplan” earlier in this chapter.
- If datalogging information is present but it does not contain report information, do you have reporting enabled? See the description of the `Log Report Information` option under “To Set the Datalogging Options for an Entire Testplan” earlier in this chapter.
- Are you sending datalogging information to the wrong directory? See the description of the `Log Directory` option under “To Set the Datalogging Options for an Entire Testplan” earlier in this chapter.

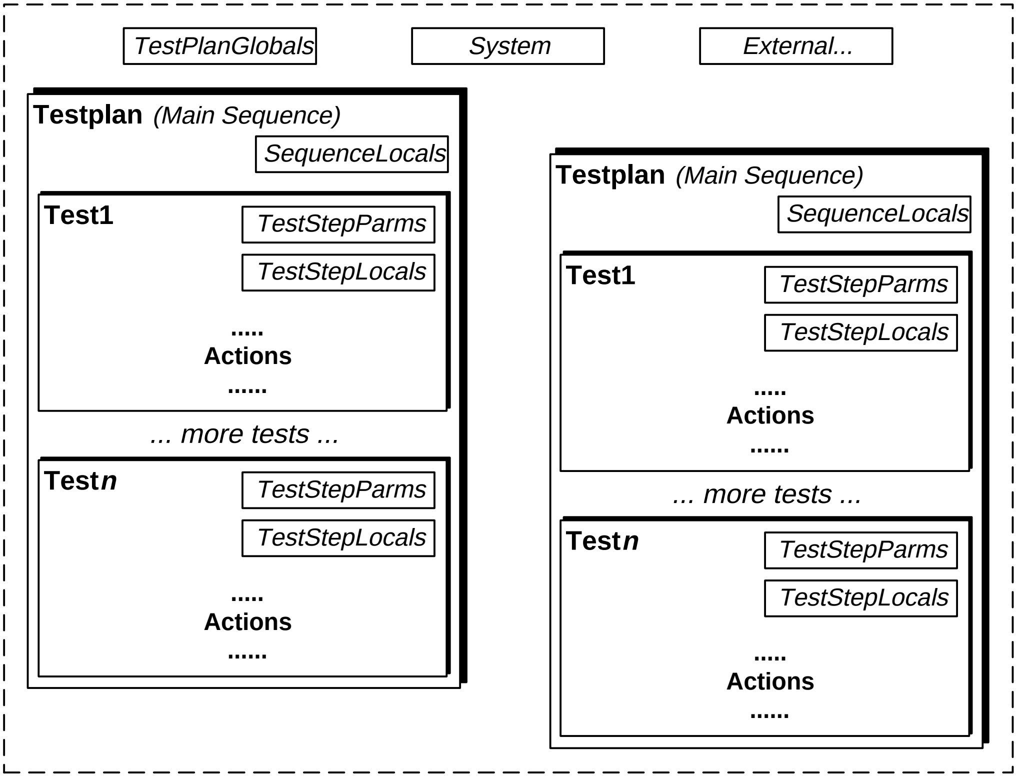
Using Symbol Tables

About Symbol Tables

Symbol tables contain data items (variables) called “symbols” whose scope makes them available within a testplan. You typically access symbols by referencing them from tests, actions, or flow control statements.

Symbols in this table...	Have this scope
SequenceLocals	Across all tests in a sequence; i.e., each sequence has its own SequenceLocals symbol table. Variables defined here can be used to pass values between tests because the variables are visible within a given sequence throughout the testplan.
System	Global to the testplan and all tests and actions in all sequences. Contains predefined symbols associated with the testing environment, such as the user ID, test system ID, and serial number of the UUT.
TestPlanGlobals	Global to the testplan and all tests and actions in all sequences. Variables defined here can pass values anywhere within a testplan.
TestStepLocals	Across all actions inside a test in a sequence; i.e., each test has its own TestStepLocals symbol table. Variables defined here can be used to pass values between actions inside the current test but not to actions in other tests.
TestStepParms	Specific to a test in a sequence; i.e., each test has its own TestStepParms symbol table. Variables defined here contain parameters passed to the test.
External (user-named)	Global to the testplan and all tests and actions in all sequences.

The hierarchy of symbol tables, and their scope, is shown graphically below.



Within the scope of testplans and tests, you can use HP TestExec SL's graphical tools to access symbol tables from the Test Executive environment. For example, View | Symbol Tables lets you examine or modify the contents of symbol tables. If you wish to interact with symbol tables from actions, you must use the C Action Development API described in Chapter 2 of the *Reference* book.

Predefined Symbols in the System Symbol Table

The System symbol table contains the following predefined symbols, all of which allow read/write access. The values of some symbols are automatically updated by HP TestExec SL, while others are simply placeholders reserved for your use; i.e., you must explicitly write values to them.

FixtureID
A string that contains a unique identifier for whichever fixture (if any) the current testplan uses to test the UUT. (placeholder)

ModuleType
A string that contains the identifier of the type of UUT. (placeholder)

OperatorName
A string that contains the name of the current login. (automatically updated)

RunCount
An Int32 whose value contains how many times the current testplan has been run since it was loaded. It starts at 1 and increments by 1 each time. Choosing a different variant does not affect its value. (automatically updated)

SerialNumber
A string whose value contains the serial number of the module currently being tested. (placeholder)

TestInfoCode
An Int32 whose value contains the code number set by the user fail mechanism. Its value is 0 if the test has not failed via the user fail mechanism. (placeholder)

TestInfoString
A string returned from the user fail mechanism. (placeholder)

TestStationID
A string that contains the identifier of a test station if you have more than one. (placeholder)

TestStatus
An Int32 whose value contains the pass/fail judgment for the most recent test. Its value is -1 if the test had no limits checker, 0 if the test passed, and *N* if the test failed. In the case of a failure, *N* equals the number of failures. For a scalar value that fails, *N* will be 1. For a one-dimensional array that fails, *N* will be the number of failing values in the array. (automatically updated)

UnhandledError A string array that contains the contents of the exception stack if an exception was detected while running a testplan. The array’s contents are the exception strings that appear in the Report window. (automatically updated)

UnhandledErrorSource A string that contains the name of the test that was executing when the most recent exception was detected. If no test was executing, the value is a null string. (automatically updated)

If desired, you can have actions in your tests examine or modify these values as needed. For example, you could examine `TestStatus` to determine if a test failed and then change the test’s parameters and rerun it before deciding that it ultimately fails. Or, you could examine the value of `RunCount` and have a test execute the first time a testplan runs but not during subsequent runs.

How Symbols Are Defined in Flow Control Statements

Be aware that symbols are defined “on the fly” when you use flow control statements. For example, specifying “For Counter = 1 to 5 Step 1” automatically creates a symbol named `Counter` in the SequenceLocals symbol table for the current sequence. As with symbols you define explicitly, you can interact programmatically with these symbols.

Note If you delete a flow control statement for which a symbol was created automatically, you must manually delete that symbol from the symbol table in which it resides.

Programmatically Interacting with Symbols

The method used to examine or modify symbols depends upon where you are when you access them.

<u>**In...**</u> — <u>**You can...**</u>

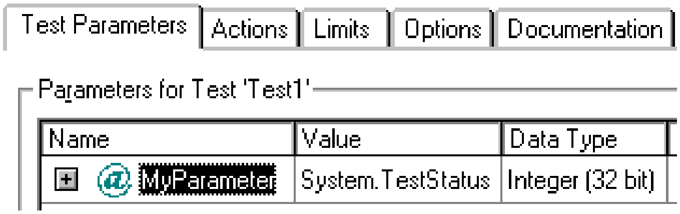
testplans

Use a flow control statement to examine or modify the value of a symbol and act upon it. The syntax for accessing symbols from flow control statements is <*symbol table*. *symbol*>. For example:

```
If System.RunCount = 1 Then
   ! Execute first time testplan runs
   test MyTest
end if
```

tests/test groups

Pass a parameter that references a symbol in a symbol table; e.g., `System.TestStatus`. (The ampersand to the left of the name of the parameter provides a visual cue that the parameter references a symbol.)



Parameters for Test 'Test1'

Name	Value	Data Type
⊞ @ MyParameter	System.TestStatus	Integer (32 bit)

actions

Use the `UtaTableRegFindData()` API function to return the value of a symbol in a symbol table.

To Examine the Symbols in a Symbol Table

1. With a testplan loaded, choose View | Symbol Tables in the menu bar.

2. When the Symbol Tables box appears, click the name of the desired symbol table in the list near Tables.

3. Browse the list of symbols and their characteristics that appears.

4. When you have finished using the Symbol Tables box, choose the Cancel button.

To Add a Symbol to a Symbol Table

1. With a testplan loaded, choose View | Symbol Tables in the menu bar.

2. When the Symbols Table box appears, click the name of the desired symbol table in the list near Tables.

3. Do the following for each symbol you wish to add:

    a. Choose the Add Symbol button.

    b. Define the characteristics of the new symbol.

       You can click a row under Value and choose the ... button to invoke a separate editor for the symbol or expand the tree view of the symbol's characteristics and edit items directly.

4. Choose the OK button.

To Modify a Symbol in a Symbol Table

1. With a testplan loaded, choose View | Symbol Tables in the menu bar.

2. When the Symbols Table box appears, click the desired symbol in the list under Symbols.

3. Modify the symbol's characteristics.

    You can click a row under Value and choose the ... button to invoke a separate editor for the symbol or expand the tree view of the symbol's characteristics and edit items directly.

4. Choose the OK button.

To Delete a Symbol from a Symbol Table

1. With a testplan loaded, choose View | Symbol Tables in the menu bar.

2. When the Symbols Tables box appears, click the symbol to be removed from the list under Symbols.

3. Choose the Delete Symbol button.

4. Choose the OK button.

Using External Symbol Tables

External symbol tables are user-defined and named symbol tables stored in a file external to the testplan. Each testplan can be associated with one or more external symbol tables, and each external symbol table can be associated with one or more testplans.

To Create an External Symbol Table

1. Choose File | New in the menu bar.

2. When prompted for which kind of document to create, choose Symbol Table.

3. Choose the OK button.

4. Do the following for each symbol you wish to define in the external symbol table:

    a. Choose the Insert button.

    b. Define the characteristics of the new symbol.

    You can click a row under Value and choose the ... button to invoke a separate editor for the symbol or expand the tree view of the symbol’s characteristics and edit items directly.

5. Choose File | Save As in the menu bar.

6. Specify the name of the file in which to store the external symbol table.

Note Although the name of the file and the name of the symbol table need not be the same, naming them alike simplifies remembering their relationship later.

7. Choose the Save button.

To Link to an External Symbol Table

To make an external symbol table visible to a testplan, you must link or associate it with the testplan. After you have linked an external symbol table to a testplan, you can use its symbols the same way you use symbols in internal symbol tables.

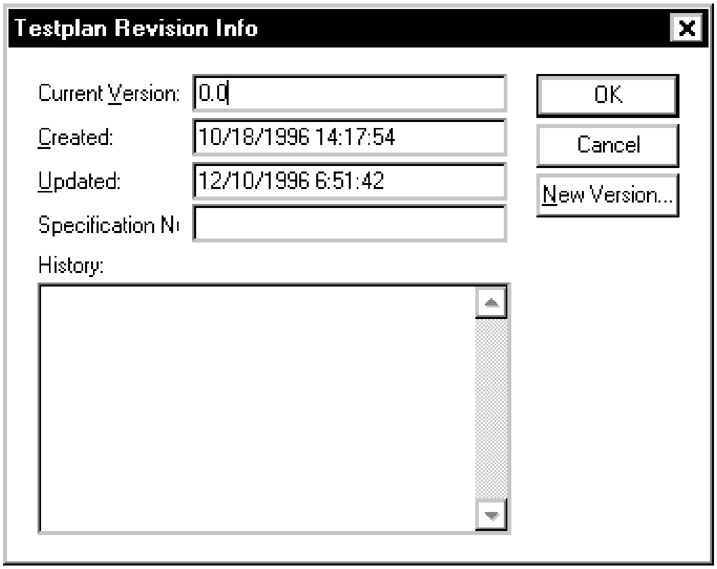
1. Choose View | Symbol Tables in the menu bar.
2. In the Symbol Tables box, choose the Link to External Symbol Table button.
3. When prompted, specify the name of the external symbol table to be associated with the testplan.
4. Choose the Open button.
5. Choose the OK button.

To Remove a Link to an External Symbol Table

1. Choose View | Symbol Tables in the menu bar.
2. In the Symbol Tables box, click the name of an external symbol table in the list near Tables.
3. Choose the Remove Link to Symbol Table button.
4. Choose the OK button.

Using Auditing

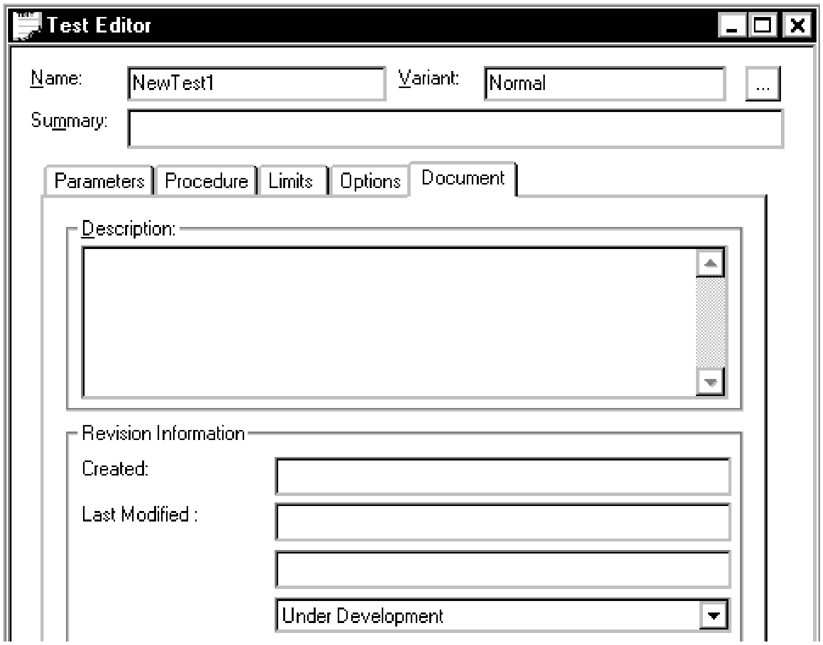
HP TestExec SL's auditing features let you document the history of software revisions as you work with the software. You can describe changes to testplans, tests, actions, and topology information. Shown below is a typical dialog box in which you can record revision information for a testplan, action, or switching topology.



Testplan Revision Info

Current Version: 0.0
Created: 10/18/1996 14:17:54
Updated: 12/10/1996 6:51:42
Specification N

As shown below, the Document tab in the Test Editor window lets you specify revision information for tests.



Name: NewTest1

Variant: Normal

Under Development

After you have entered revision information, you can view or print it the same as you do other information associated with a testplan.

Some of the auditing features are customizable; see “Setting Up the Auditing Features” in Chapter 6.

To Document Testplans, Actions & Switching Topology

1. While editing a testplan, action definition, or switching topology layer, choose File | Revision Information in the menu bar.

2. Enter a description of the current revision of the testplan, action, or switching topology.

   Tip: Use the New Version button to create a new revision when editing a testplan.

3. Choose the OK button.

To Document Tests

1. In the right pane of the Testplan Editor window, choose the Documentation tab.

2. In the data entry fields on the Documentation tab, enter a description of the current revision of the test.

3. Click to the right of the drop-down list to specify the current status of the test.

Note If you wish to customize the options that appear in this list, see “Setting Up the Auditing Features” in Chapter 6.

4. Choose the Apply button to save the description.

   Tip: If you change your mind, choose the Restore button to recall the previous description.

To View or Print Auditing Information

1. With a testplan loaded, choose View | Listing | Audit in the menu bar.

2. If you wish to print the information, choose File | Print in the menu bar while viewing the listing.

6

System Administration

This chapter provides information about configuring and administrating HP TestExec SL, which includes setting system security.

System Setup

Specifying the Location of the System Topology Layer

If you wish to use HP TestExec SL's graphical features, such as the Switching Path Editor, to control switching paths from tests, your test system must have a system topology layer defined for it. The pathname of the file containing the system layer is listed in the `[Switching]` section of HP TestExec SL's initialization file, which is "<*HP TestExec SL home*>\bin\tstexcsl.ini", as shown below.

```
[Switching]
; System topology file.
; The entry contains the name of system topology file and/or path.
; The default path is the current working directory.
System Layer=$ROOT$\bin\system.ust
```

Use a text editor, such as WordPad in its text mode, to modify this path as needed.

For an overview of controlling switching and switching topology, see Chapter 3 in the *Getting Started* book. For detailed information, see Chapter 4 in this book.

Specifying the Default Variant for a New Testplan

An entry in the [Process] section of the "<*HP TestExec SL home*>\bin\tstexcsl.ini" initialization file lets you specify which testplan variant is used as the default when you create a new testplan. The entry looks like this:

```
Default Variant=Normal
```

Use a text editor, such as WordPad in its text mode, to modify this entry as needed.

Setting Up an Operator or Automation Interface

Overview

Your goal in setting up an operator interface is to have the operator interface appear instead of the Test Executive environment used to develop testplans. Automation interfaces are similar except that they also must automatically log in to HP TestExec SL and load and run a testplan.

The methods for achieving these goals vary depending upon which language was used to develop the operator/automation interface. Interfaces developed in Visual C++ reside in a DLL that is called by HP TestExec SL, while interfaces developed in Visual Basic are external programs that call HP TestExec SL.

For information about creating an automation interface, see Chapter 1 in the *Customizing HP TestExec SL* book.

Setting Up an Automation Interface to Start Automatically

Starting an Automation Interface Created in Visual Basic

All you need to do to start an automation interface created in Visual Basic is run its executable file. Code in the automation interface handles tasks like logging in to HP TestExec SL and loading and running a testplan.

Starting an Automation Interface Created in Visual C++

Starting an automation interface created in Visual C++ requires HP TestExec SL to log in a user automatically and then load a custom user interface that supports automation tasks. When setting up an automation interface, you need to examine or edit entries in the [Process] and [Components] sections of the "<*HP TestExec SL home*>\bin\tstexcsl.ini" initialization file.

The [Process] section of the initialization file contains entries that look like this:

```
Automation=Yes
Automation User Name=<user name>
```

Setting `Automation=Yes` causes HP TestExec SL to use an automated login sequence. The user name specified for `Automation User Name` will be used during the automated login sequence. This name must belong to only one group of users, and it must not have a password associated with it.

The [Components] section of the file has an entry that follows this format:

```
<group name>=<automation DLL>
```

For `group name`, specify the group to which the user name specified for `Automation User Name` belongs. For `automation DLL`, specify the name of the DLL that contains the code for your automation interface.

Suppose a login named `AutomationUser` had no password and was the only member of a group called `Automation`. The automation-related entries for it might look like this:

```
[Process]
Automation=Yes
Automation User Name=AutomationUser
[Component]
Automation=$ROOT$\bin\stdoper.dll
```

Use a text editor, such as WordPad in its text mode, to modify these entries as needed.

For more information about specifying HP TestExec SL's security features, see “Controlling System Security.”

Setting Up Automatic Printing of Failure Tickets

If you implement a failure ticket printing scheme, you can add the following line to the [Process] section to have a failure ticket printed to the default printer:

```
Auto Print Failure Report=Yes
```

Use a text editor, such as WordPad in its text mode, to modify this entry as needed.

Specifying the Polling Interval for Hardware Handlers

If you are using a hardware handler to monitor the status of hardware, as described under “Monitoring the Status of Hardware” in Chapter 2 of the *Customizing HP TestExec SL* book, you may want to specify how frequently HP TestExec SL calls the `AdviseMonitor()` function in hardware handlers. By default, this function is called every 100 milliseconds. You can change the interval for polling by adding an entry named `Monitor Time Slice` to file “<*HP TestExec SL home*>\tstexcsl.ini” and specifying a different value in *micro*seconds, as shown below:

```
[Process]
Monitor Time Slice=500000
```

Use a text editor, such as WordPad in its text mode, to modify this value as needed.

Note The value of `Monitor Time Slice` affects the performance of your test system. The lower the value—i.e., the more frequently HP TestExec SL calls the `AdviseMonitor()` function in hardware handlers—the more time your system spends polling instead of testing.

Setting Up the Auditing Features

If desired, you can modify the behavior of some aspects of HP TestExec SL’s auditing features.

For general information about auditing features, see the auditing topics in Chapter 5.

Controlling the Appearance of the Status List

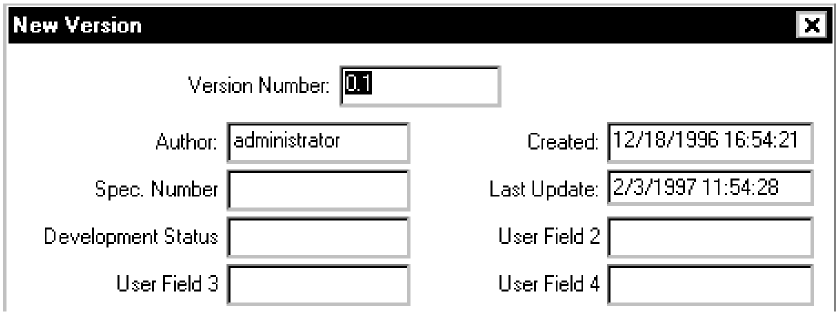
Entries in the “<*HP TestExec SL home*>\bin\tstexcsl.ini” file determine what appears in the drop-down status list on the Document tab in the Test Editor window. The default entries are:

```
[Customized Development Status]
Status1=Definition
Status2=Under Development
Status3=Under Testing
Status4=Broken
Status5=Released
```

You can change the status list’s contents by using a text editor, such as WordPad in its text mode, to modify these entries. For example, you can rename existing items and add or delete items to change the length of the list.

Controlling the Operation of the Revision Editor

If desired, you can customize some features of the New Version box (shown below) that appears when you use HP TestExec SL's auditing features to create a new version of a testplan.



New Version

Version Number: 0.1

Author: administrator

Created: 12/18/1996 16:54:21

Spec. Number

Last Update: 2/3/1997 11:54:28

Development Status

User Field 2

User Field 3

User Field 4

You can:

- Optionally prevent system operators from modifying the testplan's revision history.
- Optionally have the version number incremented automatically each time you create a new revision.
- Customize several labels associated with descriptive information entered for revisions of the testplan.

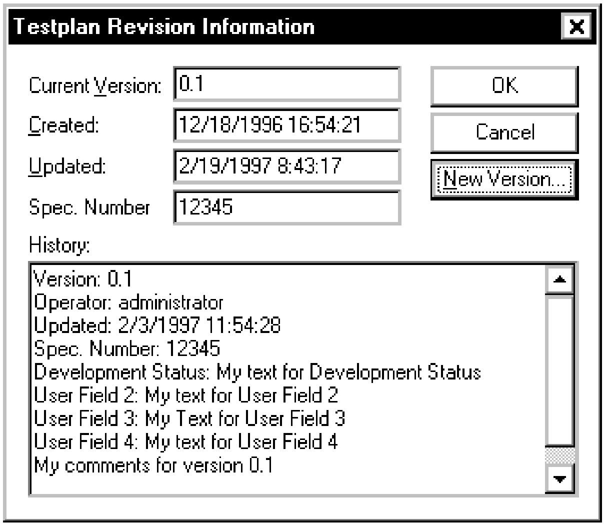
Entries in the "<*HP TestExec SL home*>\bin\tstexcsl.ini" file determine the behavior of the New Version box. The default entries are:

```
[Customized Revision Options]
Allow Operator Edit=TRUE
Automatically Increment Revision Number=TRUE
Audit Label=Spec. Number
User1 Label=Development Status
User2 Label=User Field 2
User3 Label=User Field 3
User4 Label=User Field 4
```

You can use a text editor, such as WordPad in its text mode, to modify these entries, as described below.

This entry. . .	Does this
Allow Operator Edit	When set to FALSE, prevents operators from modifying the revision history of a testplan.
Automatically Increment Revision Number	When set to TRUE, automatically increments the Version number each time new revision information is entered.
Audit Label	A label that will be associated with each new revision that is created.
User1-4 Label	Text associated with user-defined labels. This text appears in the revision history for each new version that is created.

When you enter text in the fields adjacent to the Audit Label and User1-4 Label in the New Version box, the labels and the contents of the fields appear in the revision history information displayed in the Testplan Revision Information box, as shown below. By customizing the labels, you can make them meaningful for your testing environment.



Testplan Revision Information

Current Version: 0.1
Created: 12/18/1996 16:54:21
Updated: 2/19/1997 8:43:17
Spec. Number 12345

History:

Version: 0.1
Operator: administrator
Updated: 2/3/1997 11:54:28
Spec. Number: 12345
Development Status: My text for Development Status
User Field 2: My text for User Field 2
User Field 3: My Text for User Field 3
User Field 4: My text for User Field 4
My comments for version 0.1

Directories and Files

This section lists standard directories, files, and file extensions. It also offers suggestions for locating libraries and managing temporary files.

Standard Directories

HP TestExec SL has the following standard directories and files:

HP TestExec SL
The default home directory for HP TestExec SL's files (unless you chose a different location when installing HP TestExec SL). This directory contains as subdirectories all of the standard HP TestExec SL directories listed below.

actions
Contains the definitions for some predefined actions.

bin
Contains the HP TestExec SL program and standard DLLs. Also contains the standard initialization ("*.ini") files used by HP TestExec SL.

doc
Can contain supplemental documentation.

DefaultConfiguration
Contains default copies of various files, such as initialization files.

include
Contains C header files needed by Visual C++ to build user-defined actions.

lib
Contains libraries needed by Visual C++ to build user-defined actions.

opui
Contains source code for the sample operator interface created in Visual C++

samples
Contains subdirectories that contain examples provided on an as-is basis.

Standard File Extensions

Various aspects of HP TestExec SL have associated files that are denoted by specific extensions. As noted below, some of these are registered file types that you can open simply by clicking them in the Windows Explorer.

Testplans

.tpa (For example, “testplan1.tpa”) Registered to open with HP TestExec SL.

Test libraries

.utd (For example, “arb2dmm.utd”.)

Switching topology files

.ust (For example, “system.ust”, “myfix.ust”, “myuut.ust”) Registered to open with HP TestExec SL.

HP TestExec SL supports three topology layers: system (one per test system), fixture (one per fixture type), and UUT (one per UUT type). The system “ust” file loads when HP TestExec SL starts, based on a path in the “tstexcsl.ini” file. The fixture and UUT “.ust” files (as specified under View | Switching Topology Files) load with each testplan. The system “.ust” file reloads at this time.

Actions

.umd (For example, “measv.umd”) Registered to open with HP TestExec SL.

Actions consist of a “.umd” definition file and an associated file that contains the action code. The file that contains action code can contain code for more than one action.

External symbol tables

.sym (For example, “MyTestplan.sym”) Registered to open with HP TestExec SL if no previously installed application used the “.sym” type.

Each testplan can have one or more external symbol tables associated with it.

Initialization files

.ini (For example, “tstexcsl.ini”.)

See “Initialization Files” below.

Initialization Files

HP TestExec SL has the following initialization (“*.ini”) files:

tstexcsl.ini Contains paths to other files required by HP TestExec SL and values for various system parameters. This file, which is located in the “<*HP TestExec SL home*>\bin” directory, contains comments that describe its contents.

HP TestExec SL finds this file by locating its path in file “win.ini” (located in the “<*Windows home*>” directory) under the heading “[HP TestExec SL]”. If “win.ini” does not have a specific entry for “tstexcsl.ini”, HP TestExec SL looks in the directory specified by the “windir” environment variable.

***def.ini** Various files that define datalogging formats. The Datalogging Configuration Editor provides a graphical interface for manipulating these files without your having to understand their contents.

Recommended Locations for Files

C actions (during development)
During development, keep action definitions (“*.umd”) and simple testplans to exercise them (such as “tryit.tpa”) in the same directory as the Visual C++ project used to create the action DLL (“<*action_name*>.dll”).

C actions (when ready for general use)
Action definitions (“*.umd” files) belong in directories with other logically related actions. The action DLL (“<*action_name*>.dll”) belongs in a directory specified in the PATH environment variable. See “Using Test & Action Libraries” in Chapter 5.

Test libraries
Standard test template (“*.utd”) files. You can choose your own location and organization for these files. See “Using Test & Action Libraries” in Chapter 5.

Testplans
Standard testplan (“*.tpa”) files. You can choose your own location and organization for these files. A good practice is to place related testplans in the same directory.

External symbol tables
Files containing external symbol tables (“*.sym”) belong in the directory containing the testplan with which they are associated.

Switching Topology files
Because the fixture and UUT topology layers (“uut.ust” and “fixture.ust”) are also loaded with the testplan, you should keep these files in the same directory as the testplan.

The system topology file (“system.ust”) can be located anywhere. HP TestExec SL finds this file via a path specified in file “<*HP TestExec SL home*>\tstexcsl.ini”.

utalib.vee
For HP VEE users. Provides HP VEE functions for passing parameters back and forth between HP TestExec SL and HP VEE. You may wish to move this library from the “<*HP TestExec SL home*>\lib” directory to a directory of your choice (typically the “lib” subdirectory of the “vee” installation directory).

uta.llb

For National Instruments LabVIEW users. Provides functions for passing parameters back and forth between HP TestExec SL and National Instruments LabVIEW. You may wish to move this library from the "<*HP TestExec SL home*>\lib" directory to a subdirectory called "uta.lib" in the directory where National Instruments LabVIEW is installed.

User files, DLLs & directories

You should create your own directory structure for any actions, DLLs, testplans, test libraries, and so on. See the example below.

Note: If you place your own actions, DLLs, and such in the directories created by HP TestExec SL, they may be overwritten when you install new versions of the software.

The following example illustrates a possible directory structure for customized HP TestExec SL files. Note that the "bin" directory must appear in the list of search paths for DLLs so the system can find the DLL files when executing them; see "Managing DLLs" for more information about specifying the search path for DLLs.

```
custom\
  bin\ (customized DLLs)
    eec1.dll
    abs4.dll
    autoui.dll
  actions\
    eec1\
      injpul.umd
    abs4\
      serialin.umd
  projects\ (for action sources that you create)
  testplan\
  tests\
```

Managing DLLs

While DLLs make possible much of the technology found in HP TestExec SL, they also can complicate the initial development of action code, particularly during the debugging phase. For example, it is easy to attempt to execute—with adverse results—a DLL that is not matched with a particular version of the HP TestExec SL software. It also is easy to become confused about exactly which DLL has been loaded when there are multiple copies of that DLL on a system, as frequently is the case when debugging.

How HP TestExec SL Searches for DLLs

HP TestExec SL has a specific way of looking for DLLs requested by an application that is running. It searches for them in the order listed below.

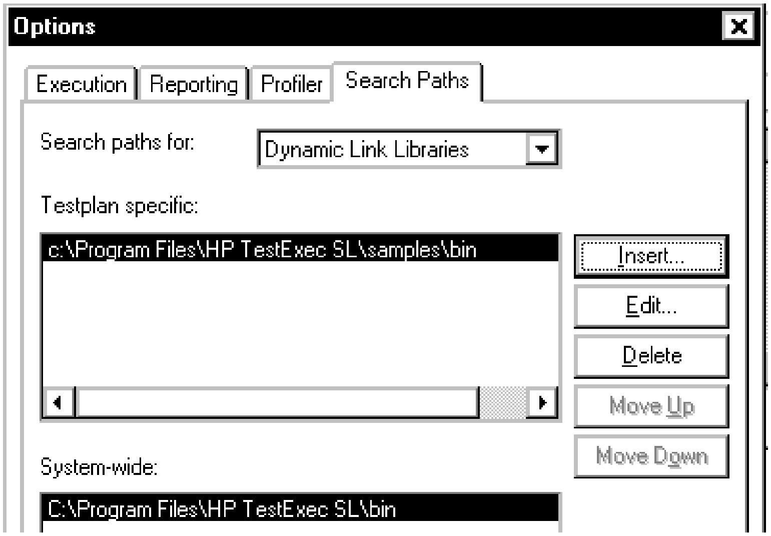
1. Use whichever DLL already is in memory.

2. If the name of the DLL is preceded by a fully qualified path, use the full pathname for the search.

   Example of a full pathname: "<*drive*>:\<*dirname*>\...\<*filename*.dll>"

3. If the name of the DLL is simply <*filename*> or <*filename*.dll>, search in this order:

   a. Search the list of paths specified for Dynamic Link Libraries in the Search Paths tab in the Options box.



Search paths for: Dynamic Link Libraries

Testplan specific:

c:\Program Files\HP TestExec SL\samples\bin

System-wide:

C:\Program Files\HP TestExec SL\bin

Testplan-specific paths are searched first, followed by System-wide paths. Both are searched in the order in which the search paths appear in their respective lists.

For more information about how to specify these search paths, see "Specifying the Search Path for Libraries" in Chapter 5.

b. Search the directory that contains the ".exe" file that is executing.

If the pathname of the DLL includes a relative path—e.g., "\<*filename.ext*>" or "<*dir*>\<*filename.ext*>—the name of the DLL is appended to the name of the directory containing the ".exe" file and that becomes the pathname for the search. For example, if the pathname of the DLL is "test\test.dll" and the ".exe" directory is "c:\tstexcsl\bin", the pathname for the search is "c:\tstexcsl\bin\test\test.dll". Or, if the pathname of the DLL is "\test.dll" and the ".exe" directory is "c:\tstexcsl\bin", the pathname for the search is "c:\test.dll".

Situations That Can Cause Problems With DLLs

If you use Visual C++ for debugging, it searches the current working directory—i.e., the directory where the project is and the latest DLL is stored—for the correct DLL to load. If the application—HP TestExec SL, in this case—changes the current working directory before that DLL can be loaded, then either the wrong DLL will be loaded (if one exists somewhere besides the project directory) or a "DLL not found" error will occur.

Because HP TestExec SL lets you specify a new working directory when loading a testplan, and DLLs containing action code are not loaded until the testplan is, action DLLs are susceptible to this problem. A temporary workaround is to create a new testplan first, using just the action desired so that the DLL gets loaded, and then loading the real testplan to check the action being debugged.

DLLs are not always unloaded from memory, especially if HP TestExec SL terminates abnormally. Thus, if HP TestExec SL is run again the DLL in memory will be used instead of the expected one. This can cause even more problems if the version in memory is out of date with the rest of the system.

DLLs must be consistent and compatible with the version of HP TestExec SL that calls them. The best way to ensure this is to build the DLL using the “include” and “lib” files for that version of HP TestExec SL to be sure of compatibility. It is also important to make sure the expected DLL and the expected HP TestExec SL software really got run. Examples of situations known to cause such problems include:

- Building an action DLL with one version of the HP TestExec SL software and executing it with another. Crashes can result.

  To prevent this, be sure that the Visual C++ “directories” option points to the correct version.

- Running an action DLL from C++ debug with the wrong version.

  To prevent this, be sure that the Visual C++ “debug” option points to the right version.

- Running an action DLL with a HP TestExec SL version it was not built for. Subtle differences can cause unexpected results, including crashes.

  To prevent this, be sure that the action DLL is being run by the correct version of HP TestExec SL.

- Running an action DLL that is already loaded. While this may be what you want, if you have created a new DLL you must remove the old one before the new will load. If the new DLL fails to behave as expected, such as not stopping at breakpoints, this may be the cause.

Symptoms Associated with Loading the Wrong DLL

Among the symptoms you may see if the wrong DLL (or wrong version of HP TestExec SL) has been loaded are:

- Unexplained crashes of DLLs that previously worked.

- Breakpoints set in a new DLL are never reached even though you know it has to be executing that code.

- A new entry point into an action is not found even though you just added it to the DLL.
- The action does not do functions you just added to the DLL.

Minimizing the Problems with DLLs

Do the following to minimize the problems caused by DLLs:

- When switching to a new version of HP TestExec SL, make sure the search path for DLLs includes it.
- Do not create too many copies of a DLL. The fewer the better.
- Before building a DLL for use in an action, be sure the C++ “directories” entry has pointers to the Version's files for Libraries and for Include.
- Before starting a debug run of HP TestExec SL from Visual C++, be sure the “debug” option points to the correct version of “tstexcsl.exe”.

Note You can use the `LoadLibrary()` function in the Visual C++ environment to load specific library files. See the Visual C++ documentation.

Managing Temporary Files

The most significant temporary files created by HP TestExec SL are the datalogging files, which by default are stored in directory “\logdir” but can be specified in the datalogging options for each testplan (Options | Testplan Options | Reporting). Ideally, the application software that uses the datalogging files automatically cleans up the temporary datalogging files. If no such application exists or if it cannot automatically delete files, you must manually delete datalogging files when you no longer need them.

Windows and Visual C++ sometimes create temporary (.tmp) files in a “temp” directory. These files can also consume disk space, and you may need to delete them occasionally.

Controlling System Security

This section tells you how to modify HP TestExec SL’s security settings. The security system controls access to program functions, based on “users” and “groups.” Users have log-in names and passwords and belong to groups. Access to program functions is based on the group to which a user belongs. Users that belong to the same group have the same access privileges to the system.

Group privileges are based on access to resources. Resources are generally tools, such as “Security” for using the security system or “SymVal” for working with symbol tables.

After your HP TestExec SL system has been installed, we recommend that you designate one person as the system administrator. The system administrator should change the password for the system user, and, optionally, add passwords to the operator, developer, and troubleshooter user groups.

Using the Default Security Settings

HP TestExec SL's default security settings give you security protection adequate for many work environments. In the default settings, user groups and user names are identical and passwords are not assigned to any group except the “system” group.

User Groups

The default user groups are as follows:

Operator An operator of a test system. Operators can select and run predefined testplans via an operator interface personality for the Test Executive, but they cannot access the test development personality.

Supervisor A supervisor of test operators.

Developer A developer of testplans, actions, and switching topology layers. A Developer can write and save testplans, action definitions, and switching topology layers, and has full access to the Test Executive’s test development environment except for system administration functions.

Administrator This group has full access to all functions and is usually assigned only to the system administrator.

System Resources

The system resources to which user groups have access are as follows:

Security Controls use of the security system.

Security Access Controls ability to modify the security system.

SymVal Controls use of values in symbol tables and modification of parameter values.

Group Access Privileges

The following table lists the specific access privileges to system resources for each default user group. The column on the left side lists the system resource in bold type and the access to that resource in plain type.

System Resource and Access	**Group**			
	Operator	**Supervisor**	**Developer**	**Administrator**
Security				
Read		x	x	x
Write				x
Edit User				x
Edit Group				x
New User				x
New Group				x
Set Access				x
Security Access				
Modify Resources				x
SymVal				
Read		x	x	x
Write				x
Print			x	x
Secure				x
Print Value				x
Edit Value			x	x
Revision				
Edit				x
Add				x

Customizing Security Settings

You can change security settings, such as:

- Assigning or changing passwords.
- Adding, deleting, or editing user and group privileges.
- Modifying access privileges for groups.

To Change a Password

1. Choose File | Security | Change Password in the menu bar.

2. Type the current password in the Old Password field.

3. Type the new password in the New Password field.

4. Retype the new password in the Confirm Password field.

5. Choose the OK button.

To Add a New User

1. Choose File | Security | Edit Security in the menu bar.

2. Click the New User button.

3. Specify the information for the new user.

   A user's information includes:

User Name	The name the user must type when logging in.
Full Name	The user's full name (which may be different from User Name).
Description	Information about the user.
Password	The password that the user must type when logging in.
Confirm Password	A verification of the password.
User Cannot Change Password	Click this box to prevent users from changing their own passwords.

User Inactive Click this box to deactivate the user’s access to the software but retain the account information for future use.

Groups: Member of/Not Member of Click the Add or Remove buttons as needed to specify the user’s membership in a group. For example, click the name of a group the user is not a member of, and then click the Add button to add the user to that group.

Note: New users are not automatically assigned to any group.

4. Choose the OK button.

To Modify an Existing User

1. Choose File | Security | Edit Security in the menu bar.
2. Click a user in the list under User Name.
3. Choose the Edit button.
4. Modify the information associated with a user.

   See “Adding a New User” above for a description of user information.

To Delete an Existing User

1. Choose File | Security | Edit Security in the menu bar.
2. Click a user in the list under User Name.
3. Choose the Delete button.

To Modify a User’s Privileges

Note Users derive their privileges from the group(s) in which they are members.

1. Choose File | Security | Edit Security in the menu bar.

2. Modify the privileges of the group(s) to which the user belongs or modify the user's membership in the groups.

To Add a New Group of Users

1. Choose File | Security | Edit Security in the menu bar.

2. Choose the New Group button.

3. In the Group Name field, type a name for the new group.

4. In the Description field, type a brief description of the new group.

5. Click the Add or Remove buttons as needed to specify which users belong to the new group. For example, click the name of a non-member, and then click the Add button to add that person to the group.

6. Choose the OK button.

To Modify an Existing Group of Users

1. Choose File | Security | Edit Security in the menu bar.

2. Select an existing group.

3. Choose the Edit button.

4. Make changes, as necessary.

Adding Custom Tools to HP TestExec SL

For an overview of custom tools, see “Using Custom Tools to Enhance the Environment“ in Chapter 3 of the *Getting Started* book.

Syntax for Adding Custom Tools

When you define custom tools, items that invoke them appear in a menu named Tools that otherwise does not appear in HP TestExec SL’s menu bar. The syntax for each item you add to the Tools menu is:[1]

<Tooln>=*<Title>*;*<Type>*;*<Specification>*

where

This item...	**Is...**	
Tooln	A name and unique numeric identifier (*n*) for a tool. The name of the tool must be “Tool” in the first level of the menu structure, and the name of a [*section*] in submenus. The numeric identifier’s value can be 0 through however many items appear at any given level in the Tools menu. The numbers must be in ascending order.	
Title	The title of an item as you want it to appear in the Tools menu. *Title* is ignored if *Type* is SEPARATOR.	
Type	The type of item, which can be:	
	EXE	Your tool is an executable program; i.e., its extension is “.exe”, “.com”, or “.bat”.
	DLL	Your tool is a function in a DLL.
	MENU	Creates a new submenu in the Tools menu.

[1] Note the use of semicolons (;) as delimiters between some items. If you omit items, be sure to leave the semicolons as placeholders.

SEPARATOR Creates a separator bar between items in the Tools menu

Specification A field whose contents vary with the *Type* of item.

If Type is...	**Then...**
EXE	Use this field to specify the pathname of the executable file to run; e.g. “c:\winnt\notepad.exe” or “c:\temp\myfile.bat”.
DLL	Use this field to specify the pathname of the DLL, followed by a space and the name of the function to run in the DLL; e.g., “c:\MyDLL.dll MyFunction”. If you omit the function’s name, it defaults to “execute”.
MENU	Use this field to specify the name of a new [*section*] that defines a submenu that contains a numbered list of custom tools.
SEPARATOR	Leave this field blank.

A simple example that runs WordPad might look like this:

```
[Tool]
Tool0=Run WordPad;EXE;c:\program files\accessories\wordpad.exe
```

A slightly more complex example that creates multiple tools might look like this:

```
[Tool]
Tool0=Run WordPad;EXE;c:\program files\accessories\wordpad.exe
Tool1=Copy Testplan Files to Production;EXE;c:\MyFiles\CopyFiles.bat
```

Finally, an example that creates multiple tools, contains a separator bar, and includes tools in a submenu might look like this:

```
[Tool]
Tool0=Run WordPad;EXE;c:\program files\accessories\wordpad.exe
Tool1=Run Custom Tool in DLL;DLL;c:\MyFiles\MyDLL.dll MyFunction
Tool2=;SEPARATOR;
Tool3=File Copying Utilities;MENU;CopyFilesSubmenu

[CopyFilesSubmenu]
CopyFilesSubmenu0=Copy Files to Production;EXE;c:\CopyToProduction.bat
CopyFilesSubmenu1=Copy Files to Archive;EXE;c:\CopyToArchive.bat
```

To Add Entries to the Tools Menu

1. Use a text editor, such as WordPad in its text mode, to open file “tstexcsl.ini” in the “bin” directory beneath HP TestExec SL’s home directory (which by default is “\Program Files\HP TestExec SL”).

2. Locate the [Tools] section in the file.

3. Add entries that conform to the syntax shown above.

4. Save the file and exit the editor.

5. Restart HP TestExec SL so it will reread the initialization file.

7

Working with VXI*plug&play* Drivers

This chapter provides information about using HP TestExec SL with standard VXI*plug&play* drivers for instruments.

What is VXI*plug&play*?

VXI*plug&play* is 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.

VXI*plug&play* instrument drivers are conceptually one layer above traditional instrument programming, which requires individual, low-level I/O statements in an application program that controls instruments. Instead, VXI*plug&play* drivers let you use higher-level languages to call predefined functions with names like `init` (initialize) and `reset` whose functionality may include numerous low-level I/O calls. Because these functions are written by those who know the instruments best—the instrument vendors—they are optimized to use the unique capabitities of each instrument.

Note

Your main source of information about VXI*plug&play* is the documentation provided with your instrument drivers. For example, you can look there to find information about the “include” files needed when using programs written in the C language to control instruments via VXI*plug&play* drivers.

How Do HP TestExec SL & VXI*plug&play* Work Together?

VXI*plug&play* instrument drivers are compatible with HP TestExec SL's strategy for hardware handlers. (For an overview of hardware handlers, see Chapter 3 in the *Getting Started* book.) For example, both VXI*plug&play* drivers and hardware handlers include functions to initialize, reset, and close hardware modules such as instruments.

Besides a small set of function calls shared by VXI*plug&play* drivers and HP TestExec SL's hardware handlers, each driver strategy has its own unique aspects. VXI*plug&play* drivers include functions that are specific to particular types of instruments. For example, although the drivers for a DMM and a frequency counter both provide functions to initialize and reset them, one instrument requires different functions to control it than does the other because the functionality of the instruments is dissimilar. In a similar fashion, HP TestExec SL's hardware handlers may include additional functions that are specifically used to control switching hardware—such as `SetPosition()` and `GetPosition()`—via the Switching Path Editor.

When HP TestExec SL runs, it automatically calls as needed the following functions in hardware handlers or VXI*plug&play* drivers associated with hardware modules via HP TestExec SL's Switching Topology Editor:

- Functions used to initialize hardware prior to using it.

  In hardware handlers, this is the `Init()` function (which also resets the module when called). In VXI*plug&play* drivers, these are functions whose names include "init"; e.g., `hp34401_init`.

- Functions used to reset hardware to a known state.

  In hardware handlers, this is the `Reset()` function. In VXI*plug&play* drivers, these are functions whose names include "reset"; e.g., `hp34401_reset`.

- Functions used to close—i.e., terminate communication with—hardware.

In hardware handlers, this is the `Close()` function. In VXI*plug&play* drivers, these are functions whose names include “close”; e.g., `hp34401_close`.

Besides automatically initializing, resetting, and closing instruments via VXI*plug&play* drivers, HP TestExec SL lets you interactively control instruments from action code that you write. The method for doing this is described next.

How Do Actions Control Instruments via VXI*plug&play*?

HP TestExec SL provides API functions used to communicate with instruments from actions via VXI*plug&play* drivers. Shown below is an example of how an action written in C can communicate with an instrument via a VXI*plug&play* driver.

```
void UTADLL ProgramPowerSupply (HUTAPB hParameterBlock)
{
// Action routine that programs an HP 66312 power supply.
// Example assumes that parameter block contains three parameters:
//   Voltage - type Real64
//   Current - type Real64
//   PowerSupply - type Inst

// Assign miscellaneous variables
HUTAREAL64 hData;
ViStatus ErrorCodes;
HUTAINST hInstrument;

// Get value of voltage from parameter block
hData = UtaPbGetReal64(hParameterBlock, "Voltage");
double dVolt = UtaReal64GetValue(hData);

// Get value of current from parameter block
hData = UtaPbGetReal64(hParameterBlock, "Current");
double dCurr = UtaReal64GetValue(hData);

// Get the ViSession identifier from the parameter block
hInstrument = UtaPbGetInst(hParameterBlock, "PowerSupply");
long lViSession = UtaInstGetViSession(hInstrument);

// Set the voltage & current, and turn on the output
ErrorCodes = hp66312_voltCurrOutp (lViSession, dVolt, dCurr);

...(optional code that checks ErrorCodes for power supply errors)

return;
}
```

In the example above, a call to `UtaPbGetInst()` returns the handle to a data container that contains data for an instrument—in this case, a power supply—from the action's parameter block. Given that handle, a call to `UtaInstGetViSession()` returns a unique identifier for the instrument's ViSession. Once the identifier of the ViSession is known, the example uses a standard VXI*plug&play* call, `hp66312_voltCurrOutp`, to program the power supply to voltage and current limit settings passed in as parameters—"Voltage" and "Current"—in the action's parameter block.

The same example is shown below rewritten in C++ to use data types that HP TestExec SL implements as C++ classes. Although the syntax differs somewhat from the C example, the concepts are similar.

```
void UTADLL ProgramPowerSupply (HUTAPB hParameterBlock)
{
// Action routine that programs an HP 66312 power supply.
// Example assumes that parameter block contains three parameters:
//   Voltage - type Real64
//   Current - type Real64
//   PowerSupply - type Inst

// Assign miscellaneous variables used in this function
ViStatus ErrorCodes;
long lViSession;

// Assign variables from parameters used by this action routine
IUtaInst hPowerSupply (hParameterBlock, "PowerSupply");
IUtaReal64 Volt(hParameterBlock, "Voltage");
IUtaReal64 Curr(hParameterBlock, "Current");

// Get the ViSession identifier from the instrument handle
lViSession = UtaInstGetViSession (hPowerSupply);

// Set the voltage & current, and turn on the output
ErrorCodes = hp66312_voltCurrOutp (lViSession, Volt, Curr);

...(optional code that checks ErrorCodes for power supply errors)

return;
}
```

Programming other instruments from actions via VXI*plug&play* drivers is similar. Once you have obtained the identifier of a ViSession with the instrument, you can call functions in the VXI*plug&play* driver.

For more information about creating C actions, see “Working with C Actions” in Chapter 3 of this book.

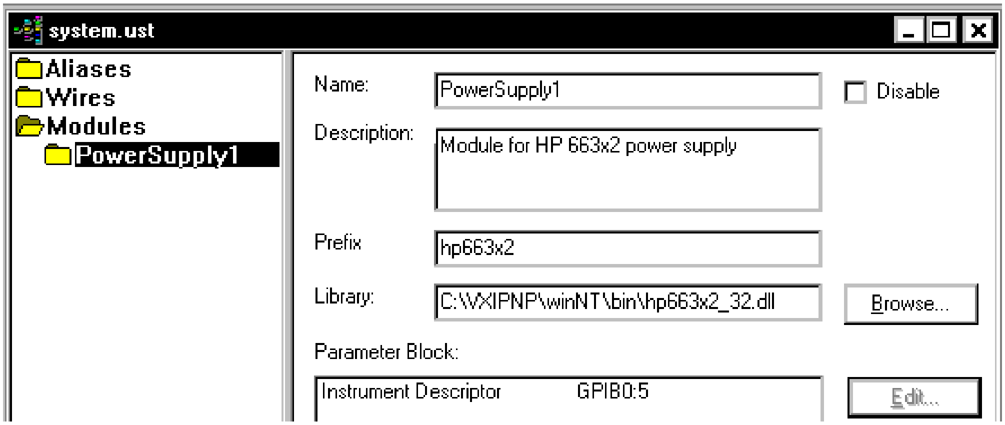
To Control a VXI*plug&play* Instrument from an Action

Configuring HP TestExec SL to Use VXI*plug&pla*y Instruments

Before an action can control instruments via VXI*plug&play*, you must make HP TestExec SL aware of those instruments, as described below.

1. If the necessary I/O libraries and VXI*plug&play* drivers for instruments are not already installed, install and configure them as described in their documentation.

2. Use HP TestExec SL's Switching Topology Editor to add to the test system's topology a module for each instrument that uses a VXI*plug&play* driver, as shown below. Typically, you will do this in the system layer of topology; i.e., in file "system.ust".



Name: PowerSupply1

□ Disable

Description: Module for HP 663x2 power supply

Prefix hp663x2

Library: C:\VXIPNP\winNT\bin\hp663x2_32.dll

Parameter Block:

Instrument Descriptor GPIB0:5

Do the following when associating an instrument that uses a VXI*plug&play* driver with a module:

- For the Prefix, enter the name of the instrument as it appears in calls to the driver; e.g., calls to HP 663x2-series power supplies begin with “hp663x2” (as in “hp663x2_init”) so that is what you should enter.

- For the Library entry, specify the name of the DLL in which the VXI*plug&play* driver for the instrument resides.

- Press the Add button to load the parameter block.

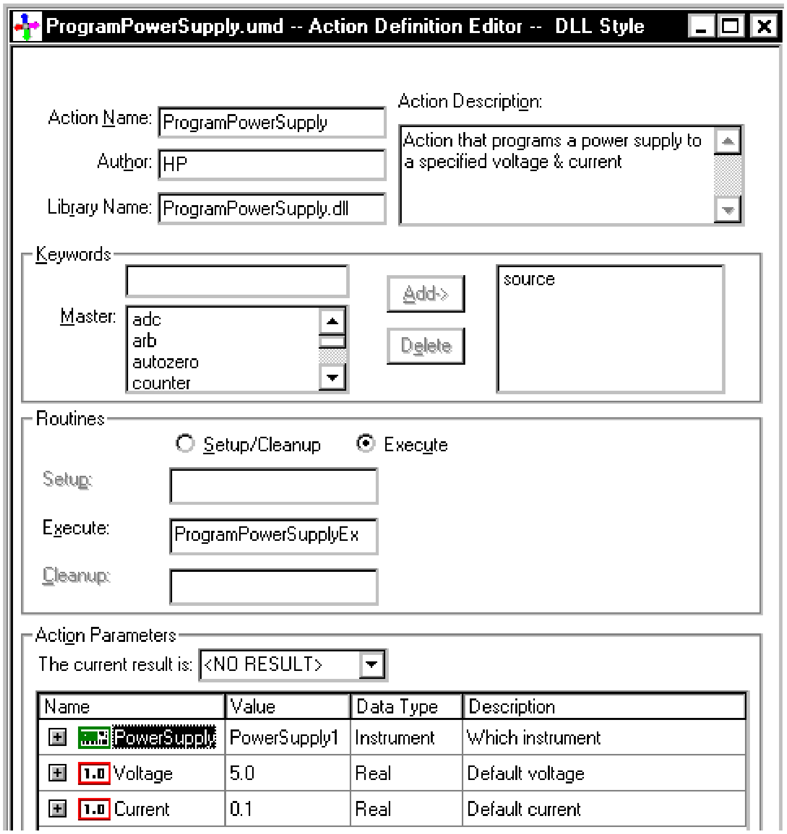
- Use the value of the Instrument Descriptor in the parameter block to define a unique instance of the instrument. For example, if your test system has two HP 663x2-series power supplies, you might give the first a Name of “PowerSupply1” at address “GPIB0::5” and add a second module named “PowerSupply2” at some other I/O address.

- Press the Update button to save your changes.

For more information about using the Switching Topology Editor, see Chapter 4 in this book.

Creating the Action

Once HP TestExec SL is aware of instruments in your test system controlled via VXI*plug&play*, you can create actions that control them. Actions that control instruments via VXI*plug&play* are similar to other kinds of actions except that when using the Action Definition Editor to define the action, you must add to the parameter block a parameter whose type is “Instrument” for the instrument you wish to control. The example below shows this parameter as well as parameters for setting the voltage and maximum current output from a power supply.



ProgramPowerSupply.umd -- Action Definition Editor -- DLL Style

Action Name: ProgramPowerSupply

Author: HP

Library Name: ProgramPowerSupply.dll

Action Description: Action that programs a power supply to a specified voltage & current

Keywords

Master: adc, arb, autozero, counter

source

Routines

○ Setup/Cleanup ● Execute

Setup:

Execute: ProgramPowerSupplyEx

Cleanup:

Action Parameters

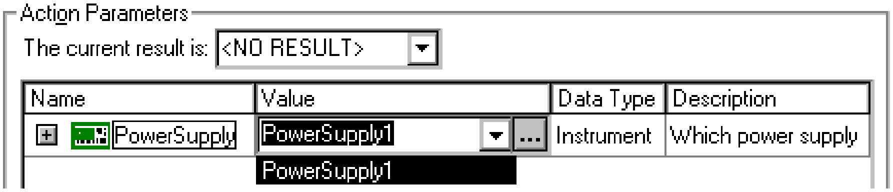
The current result is: <NO RESULT>

Name	Value	Data Type	Description
⊞ PowerSupply	PowerSupply1	Instrument	Which instrument
⊞ 1.0 Voltage	5.0	Real	Default voltage
⊞ 1.0 Current	0.1	Real	Default current

Do the following to add a parameter for the instrument:

1. In the Action Definition Editor, choose the Add button.
2. When the row containing a new parameter appears, set the Data Type to “Instrument”.
3. Enter a Name and Description for the parameter.

4. Click under Value and choose the desired instrument from the drop-down list that appears there, as shown below.



Action Parameters

The current result is: <NO RESULT>

Name	Value	Data Type	Description
⊞ PowerSupply	PowerSupply1	Instrument	Which power supply

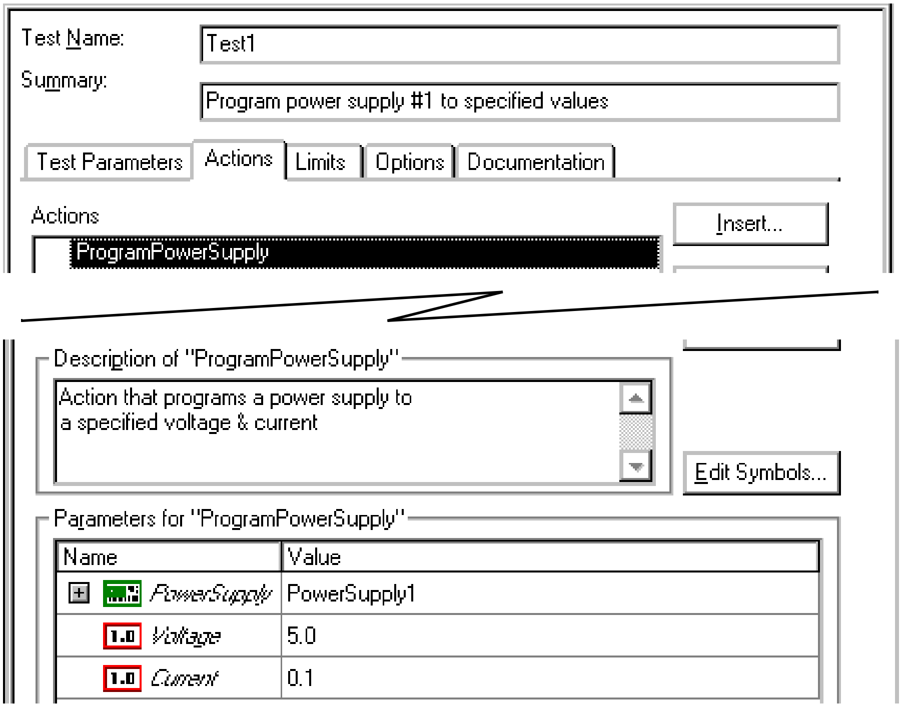
PowerSupply1

Now you must write the code that implements the action using the concepts described earlier under “How Do Actions Control Instruments via VXI*plug&play*?”

For more information about using the Action Definition Editor to define actions, see “To Define an Action“ and “Using Parameters with Actions” in Chapter 3 in this book.

Using the Action in a Test

As shown below, using an action that programs an instrument via a VXI*plug&play* driver is similar to using other kinds of actions in tests. The only thing different is that this action’s parameter list includes a parameter that identifies which instrument is being controlled.



Test Name: Test1

Summary: Program power supply #1 to specified values

Actions

ProgramPowerSupply

Description of "ProgramPowerSupply"

Action that programs a power supply to
a specified voltage & current

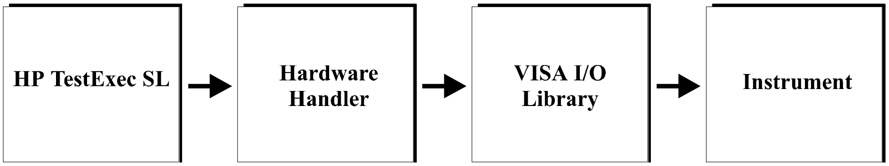
Parameters for "ProgramPowerSupply"

Name	Value
⊞ *PowerSupply*	PowerSupply1
1.0 *Voltage*	5.0
1.0 *Current*	0.1

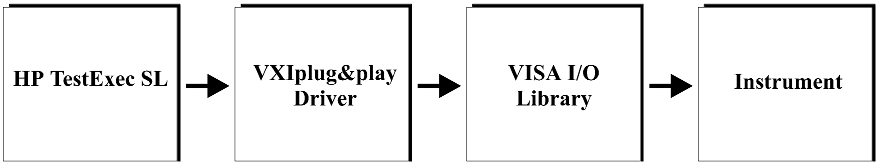
For more information about using actions in tests, see “Adding Actions to a Test/Test Group“ in Chapter 2 of this book.

Beyond VXI*plug&play*

A conceptual diagram of the layering of hardware and software when using HP TestExec SL with hardware handlers is shown below.



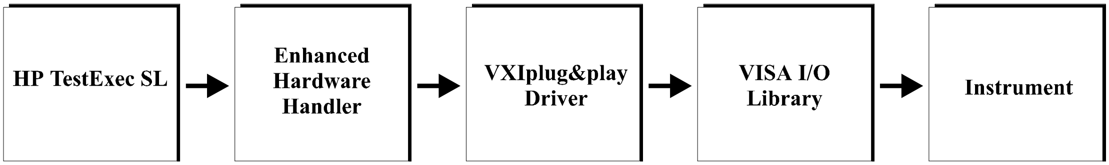
The model when using HP TestExec SL with VXI*plug&play* instruments looks like this:



Recall that hardware handlers and VXI*plug&play* drivers are compatible, but that their specific implementation—i.e., features—can vary considerably from one handler or driver to the next. For example, VXI*plug&play* drivers contain functions whose implementations are specific to a particular instrument. Although you trigger both a frequency counter and a voltmeter to take a reading, each type of instrument performs a different function and requires different commands to trigger it. Similarly, you might program each to a specific range prior to triggering it, but the details of the commands required to change ranges would be different for a counter and a voltmeter.

As with VXI*plug&play* drivers, hardware handlers provide functionality that is unique to them. For example, hardware handlers let you send status messages to HP TestExec SL's Watch window during debugging (with the `DeclareStatus()` function). Also, they let you monitor the status of tracing (with the `AdviseTrace()` function) and modify the hardware handler's behavior "on the fly" as appropriate for greater speed.

Suppose you could combine the functionality of hardware handlers and VXI*plug&play* drivers. Ideally, the combination would provide instrument-specific features needed to control instruments plus the enhanced interaction with HP TestExec SL’s features possible via hardware handlers. The conceptual diagram below shows how this is possible *without modifying the VXI*plug&play *driver.*



If desired, you can create an enhanced hardware handler that communicates with or “handles” the VXI*plug&play* driver. When HP TestExec SL calls instrument-specific functions that reside in the VXI*plug&play* driver, you have the option of passing them through the hardware handler unmodified or enhancing their behavior.

For an example of an enhanced hardware handler that adds status information to a VXI*plug&play* driver, search online help for “example, sample code for enhancing a VXIplug&play driver.” Comments in the example describe how to use the Switching Topology Editor to associate handlers/drivers with modules when using this strategy.

For more information about the features of hardware handlers, see “About Hardware Handlers“ in Chapter 3 of the *Getting Started* book. For more information about creating hardware handlers, see Chapter 2 in the *Customizing HP TestExec SL* book.

8

Using String Formatting

This chapter provides information about how you can use HP TestExec SL to create strings that contain replaceable parameters whose values are updated when the string is read from or written to.

What is a Formatted String?

HP TestExec SL lets you create or modify strings, apply formatting to strings, and have the formats stored with the strings for subsequent use. As shown below, a formatted string consists of one or more fixed parts that you specify plus one or more replaceable parameters or placeholders for which actual values are substituted when reading or writing the string.

Formatted String = Fixed Parts + Replaceable Parameters

Refer the the example below.

```
The serial number is %System.SerialNumber%
```

The string has a fixed part and a replaceable part. The fixed part, which you specify as a standard string, is "The serial number is". The replaceable part, whose actual value will be determined when the string is used, is "%System.SerialNumber%". Notice how it is enclosed in percent signs (%). This placeholder references a variable named `SerialNumber`, which is a symbol in a symbol table named `System`.

The Two Types of Formatting Operations

There are two types of formatting operations you can assign to strings in HP TestExec SL:

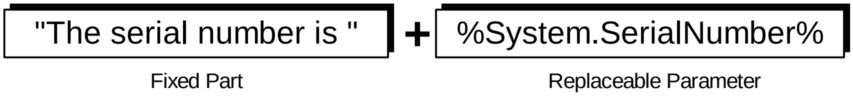
Update string from parameters The string is updated from the values of items referenced by replaceable parameters in the string. The formatting operation occurs before the value of the string is returned to the code that requests it.

Update parameters from string The values of items referenced by replaceable parameters in the string are updated from the string. The formatting operation occurs after the value of the string is written.

Each is individually described below.

Updating a String from its Replaceable Parameters

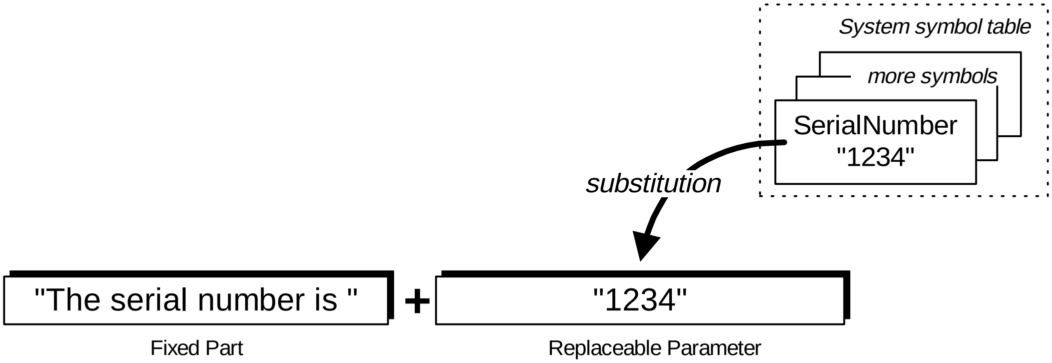
Suppose the formatting operation specified for a string is “Update string from its parameters.” A diagram of the string from the previous example prior to replacing the paremeter with its actual value is shown below.



Suppose the value of SerialNumber is “1234”. If so, the actual value returned when the string is read will be:

```
The serial number is 1234
```

The diagram below shows how the substitution of an actual value from the symbol table is made.



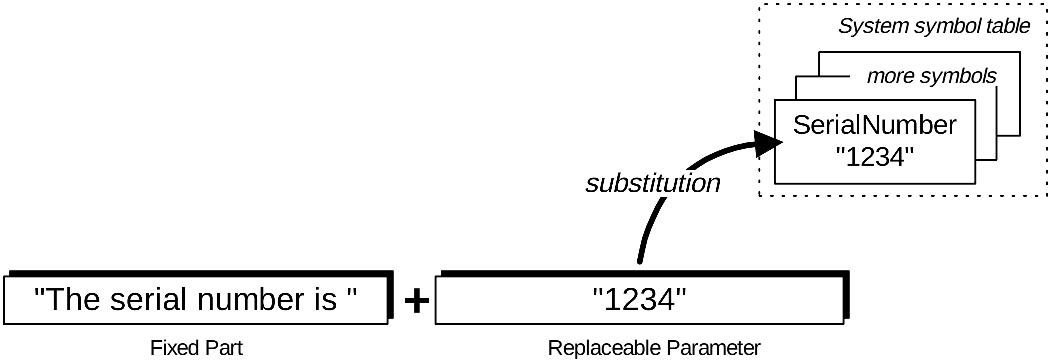
Although this example is simple, there could have been more than one replaceable parameter and they could have appeared anywhere—at the beginning of the string, in the middle of the string, at the end of the string, or in various combinations. Also, there could have been several fixed parts to the string, with replaceable parameters interspersed among them. For example, prior to replacement the string could have been:

```
The serial number is %System.SerialNumber% for the module
```

Here, the string consists of two fixed parts—”The serial number is” and “for the module”—and one replaceable parameter between them.

Updating Replaceable Parameters from a String

What happens when the formatting operation specified for a string is “update parameters from string?” The string is parsed and the values of items referenced by replaceable parameters in the string are updated from the string. Refer to the example below.



Here, the value of the replaceable parameter in the string is passed to the symbol named `SerialNumber` in the `System` symbol table. The string for this operation looks the same as in the previous example; i.e.,:

```
The serial number is 1234
```

However, this time the formatting operation is “Update parameters from string”. Because the format specifies “The serial number is” as the fixed part, followed by a replaceable parameter that references a symbol in a symbol table, the string is parsed when written to so that the actual value at the replaceable parameter is written to its associated symbol.

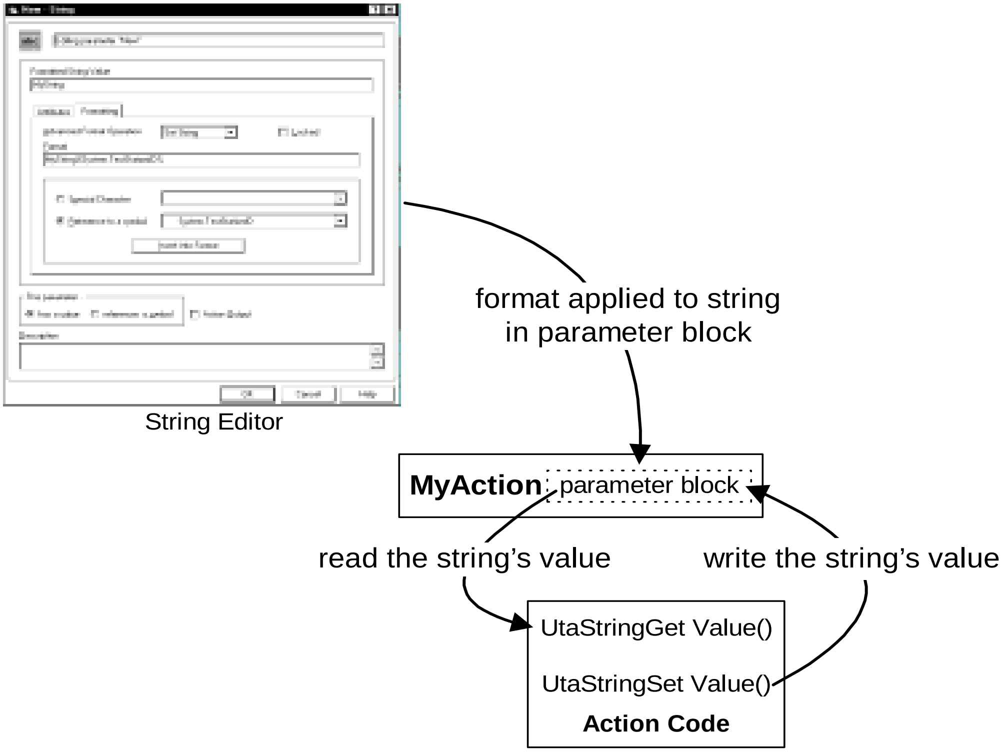
Note

If you wish to use the “Update parameters from string” formatting operation to update the values of parameters in an action’s parameter block, those parameters must be “output” parameters; i.e., their Action Output option must have been enabled when they were created with the Action Definition Editor.

How Does String Formatting Work?

String formatting lets you apply a format to a string in a parameter blocks or in a symbol table. Subsequent use of the strings makes use of the format by applying the formatting when reading from or writing to the string.

Refer to the example below. Here, the String Editor[1] was used to apply a format to a string in a parameter block in an action named “MyAction” that was created with the Action Definition Editor. `UtaPbGetString()` and `UtaPbSetString()` functions in action routines can access the string, which will be read or written with the specified format applied to it.



[1] The String Editor is one of several editors that let you edit various types of data in HP TestExec SL.

Which Data Types are Supported for Replaceable Parameters?

The following data types are supported for replaceable parameters in string formats.

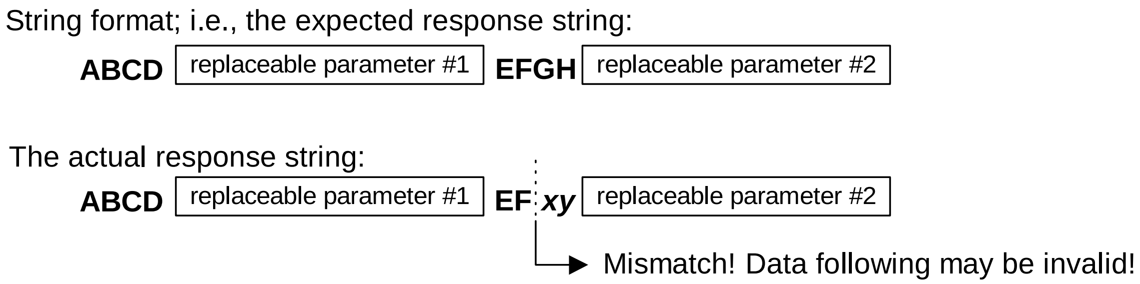
Data type	Comments
Int32	
Int32Array	One dimension only, elements are written/read separated by commas
Real64	
Real64Array	One dimension only, elements are written/read separated by commas
String	
StringArray	One dimension only, elements are written/read separated by commas
Waveform	Elements are written/read separated by commas, only values of samples—i.e., point data—are used

Note

Even though they are used in strings, replaceable parameters do not necessarily need to be of type String. Because they are surrounded by percent signs—e.g., %MyParm%—that have a special meaning to HP TestExec SL, replaceable parameters are automatically translated between their native data types and strings as needed.

What Happens if “Update Parameters from String” Fails?

Correct operation of string formatting when updating replaceable parameters from a string assumes that values in the actual string match the values that you specify in the format. For example, with the exception of spaces in a formatted string, characters in the fixed part of the string format and the actual string must match exactly.[1] If they do not match, any values beyond the point of the mismatch when the string is read sequentially are suspect, as shown below.


String format; i.e., the expected response string:
ABCD
replaceable parameter #1
EFGH
replaceable parameter #2
The actual response string:
ABCD
replaceable parameter #1
EF
xy
replaceable parameter #2
Mismatch! Data following may be invalid!

If a mismatch occurs, the values returned for replaceable parameters beyond the point of the mismach are error values based upon the data type of the replaceable parameters, as shown below.

If the data type of the replaceable parameter is...[a]	The returned error value will be...
Int32, Int32Array	-99999999
Real64, Real64Array, Waveform	-9.99e37
String, StringArray	“ScanError”

a. Arrays used with string formatting must be single-dimensional.

[1] A single space (blank or tab) in the format matches an arbitrary number of space characters in the string value, including zero space characters.

Note Keep in mind this need for matching formats when working with strings used with instruments that are IEEE-488.2 compliant. The commands used to program those instruments use commas and semicolons as separators between values, and so should string formats used with them. Otherwise, values cannot reliably be distinguished from one another.

If the formatting operation is “update parameters from string” and the replaceable parameter in a string is an array data type, the scanned value does not need to provide values for all the elements in the array. Any elements for which values are not explicitly specified will be set to the values listed below.

If the data type of the element is...[a]	The value will be set to...
Int32Array	99999999
Real64Array, Waveform	9.99e37
StringArray	“NoValue”

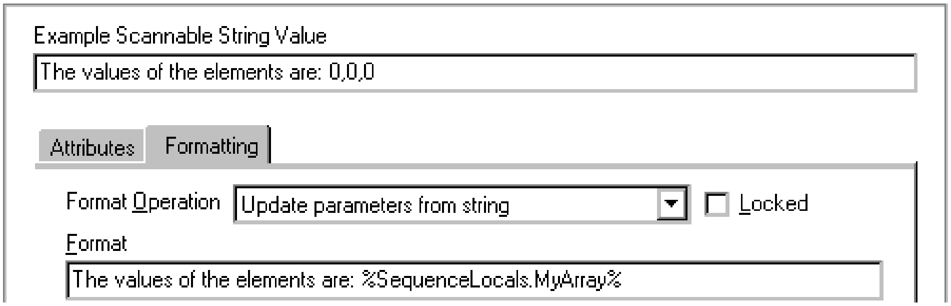
[a] Arrays used with string formatting must be single-dimensional.

Notes About String Formatting

Keep the following limitations in mind when using the “update parameters from string” formatting operation.

- Scanning a string for replaceable parameters whose data type is Int32 returns all adjacent numeric values. This means that two Int32 parameters in a row must be separated by a fixed part. If not, both values will be merged to return the value of the first replaceable parameter and the second replaceable parameter will return an error value. Similarly, a Real64 parameter immediately followed by an Int32 parameter—i.e., with no fixed part separating the values—will cause a scanning error for the Int32 parameter.

- The values for elements in an array must be separated by commas when they appear in formatted strings used to update parameters. Consider the following formatting operation, which references a single-dimensional array named `MyArray` in the `SequenceLocals` symbol table:



Example Scannable String Value
The values of the elements are: 0,0,0

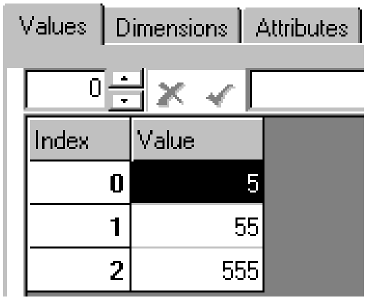
Format Operation: Update parameters from string
□ Locked
Format: The values of the elements are: %SequenceLocals.MyArray%

Suppose you used the following string in your action routine to write values to the elements in the array:

```
The values of the elements are: 5,55,555
```

Notice how commas are used as separators between values written to the array’s elements.

The array would look like this after the values were written to it:



Index	Value
0	5
1	55
2	555

- String arrays must not have commas in the values for their elements because commas are used to separate values when scanning the elements in arrays.

- If a replaceable parameter contains only the name of a symbol—i.e., no specifier for the table in which the symbol resides—the order used to search for the symbol is the current symbol table followed by all public symbol tables in the order of their scope within the testplan.

How are Formatted Strings Useful?

Serial numbers provide an example of how string formatting can be useful. Suppose the serial numbers for modules on your test system consist of a module name plus a unique identifier for each module. For example, the name of the module might be “ControlModule” and the unique identifiers for individual modules might range from 1111 to 9999. If so, a complete module serial number might be “ControlModule2345”.

The `System` symbol table contains predefined symbols named `ModuleType` and `SerialNumber`, both of which are strings. When it begins running, your testplan could prompt the system operator for the module type and write it to `ModuleType`. As it tested individual modules, your testplan could use a bar code reader to read identifiers from individual modules and temporarily store them in `SerialNumber`. If its formatting operation was specified as “Update string from parameters,” the formatted string shown below could be used in action code to concatenate the values to produce the complete serial number for each module.

```
Serial number is %System.ModuleType%%System.SerialNumber%
```

The complete serial number could be printed on failure tickets as needed or used elsewhere, such as in datalogging to collect information about the testing process.

The use of string formatting is described in greater detail with related topics. The mechanics of applying string formatting are described in the online help for HP TestExec SL’s String Editor.

9

Using Actions to Control Message-Based Instruments

This chapter provides information about using predefined actions in HP TestExec SL to control message-based instruments, which makes use of string formatting features described in Chapter 8, “Using String Formatting.”

Overview of Controlling Message-Based Instruments

Why Use Actions to Control Message-Based Instruments?

Message-based instruments are instruments controlled by human-readable messages written in a high-level programming language for instruments, such as SCPI (Standard Commands for Programming Instruments), rather than by a more cryptic method such as directly writing/reading registers in the instrument. However, even using a high-level instrument control language typically requires knowledge of the command syntax of instruments.

If you do not already have actions that control message-based instruments, features in HP TestExec SL let those who understand the syntax for controlling instruments create new actions that are readily usable by others who do not necessarily know how to program instruments. Actions used to control instruments resemble other actions but have instrument-specific parameters in which users specify data to be sent to or received from instruments. These actions are used with hardware handler software that:

- Manages and interfaces with message-based instruments
- Provides a standard set of functions that send commands and receive responses as well as reset the instrument, query its error status, and retrieve its ID. You can customize these for specific instruments.
- Captures in HP TestExec SL's Trace window the history of commands sent to and received from instruments
- Supports the display of instrument-specific state information in HP TestExec SL's Watch window
- Supports automatic creation of instrument-specific nodes for use in HP TestExec SL's switching topology

When Can I Use Actions to Control Message-Based Instruments?

You can use actions to control a message-based instrument if:

- Your interaction with the instrument is limited to replacing parameters of type Int32, Int32Array, Real64, Real64Array, String, StringArray or Waveform in commands sent to or received from the instrument
- You do not need to handle SRQs (service requests) from the instrument[1]
- The instrument is GPIB, RS-232, or VXI-based
- The instrument is message-based; i.e., it responds to messages sent in strings. Instruments that are IEEE-488.2 compliant are examples of message-based instruments.

[1] You can use simple error checking at the end of each message sent or monitor the instrument for a timeout condition.

Using Actions to Control Message-Based Instruments

Note

Topics in this section describe the use of formatted strings. For an overview of formatted strings, see Chapter 8, “Using String Formatting.”

Adding the Instrument to the Switching Topology

Do the following once for each message-based instrument you wish to control via actions.

1. Physically install the instrument in the system, such as by connecting it to GPIB or by inserting it into a VXI card cage

2. Add the instrument to the system layer of the switching topology by following the general instructions in “To Add a Module” in Chapter 4 and:

   a. Specifying “hwhmsginst.dll” as the Library

   b. Choosing which interface style the instrument uses, which can be stdGPIB, stdVXI, or stdSerial

   c. Filling in the parameters for the instrument

      Usually, this means setting the address of the instrument in parameters (logical address for VXI and GPIB). Use the parameter descriptions to guide your choices. You can let other parameters use their default values if the instrument is compliant with SCPI or IEEE-488.2, which includes most new message-based instruments. If you have more than one of the same instrument or module, the name should reflect that; e.g., the first power supply could be “PS1” and the second “PS2.”

Which Actions Does HP TestExec SL Provide?

The following actions are provided with HP TestExec SL for controlling message-based instruments. They reside in directory "<*HP TestExec SL home*>\actions". Note that their names begin with "msginst".

msginstReset	Resets the instrument
msginstGetErrorStatus	Returns the error status byte from the instrument[a]
msginstGetID	Returns an identifying string from the instrument[b]
msginstSend	Sends a command to the instrument and does not expect a response
msginstReceive	Return a response string from the instrument if one has previously been requested from the instrument via a `msginstSend` action
msginstQuery	Sends a command to the instrument and waits until it receive a response
msginstBinarySend	Sends an arbitrary byte array as part of a message to the instrument
msginstBinaryReceive	Receive an arbitrary byte array as part of a response from the instrument

[a] The action's default settings are correct for IEEE-488.2 compliant instruments. Others may need changes to the command sent to the instrument and the mask used to isolate error status bits.

[b] The action's default settings are correct for IEEE-488.2 compliant instruments to return a response that uniquely identifies the instrument.

You can find detailed descriptions of these actions by choosing the Details button on the Insert Action box before inserting them into a test.

Choosing Which Action to Use

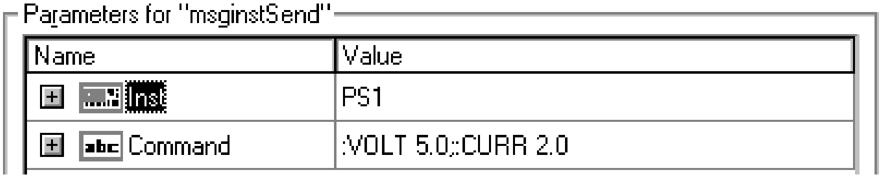
Before you can control an instrument, you must know which kinds of measurement-related tasks you wish to do with it. For example, a typical task associated with a power supply might require a "set voltage and current limit" action to program it to a known state. Once you have identified a task, choose an action that is appropriate for it. For example, for a power supply you could choose `msginstSend` to send a message that programs it to a known state.

Setting Up the Action

Begin by verifying that the action you have chosen can communicate with the desired instrument, and that you are using the correct command syntax to control that instrument, as described below.

1. Create a new testplan for setting up the action, add the chosen action to a new test in the testplan, and specify the action’s parameters.

   In the case of an action that controls a power supply, you might specify the parameter named `Inst` to reference instrument “PS1” and the parameter named “Command” to set a particular voltage and current for the power supply, such as “:VOLT 5.0;:CURR 2.0”. For example:



Parameters for "msginstSend"

Name	Value
⊞ Inst	PS1
⊞ abc Command	:VOLT 5.0;:CURR 2.0

2. Run the test and make sure the instrument is receiving the command and acting on it.

   For a power supply, you could use its front panel to examine the setttings for voltage and current, or use a multimeter to verify its output.

For more information about adding a test to a testplan, see Chapter 1. For more information about adding actions to tests and specifying their parameters, see Chapter 2.

Copying the Action Definition

Because you do not want to overwrite the original action definition, which you may need to use again, you must create a copy of it for your use. Make a copy of the action definition by:

1. Selecting the action in the list under Actions.

2. Choosing the Details button to invoke the Action Definition Editor.

3. Choosing File | Save As in the menu bar.

4. Specifying a new, unique name for the action definition.

   Use a name that reflects the task the action does, and place it in a directory that HP TestExecSL searches for action definitions. For example, the name might consist of a descriptor for the instrument followed by whatever task the action does, such as “HP6632ProgramVI.”

Note

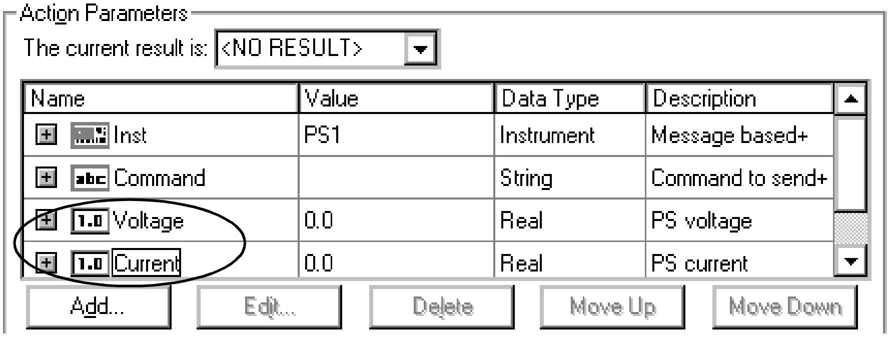
If you accidentally overwrite an original action definition, look in directory “<*HP TestExec SL home*>\samples\msginst\MsgInstUmd” for a copy of it.

Customizing the Action Definition

Now you can customize the generic action definition you copied so that it is usable by those who do not understand the commands used to program instruments. Do the following:

1. Use the Add button in the Action Definition Editor to add task-specific parameters to the copy of the action definition.

   For an action that programs a power supply, you might add two new parameters named “Voltage” and “Current”, as shown below.



Action Parameters

The current result is: <NO RESULT>

Name	Value	Data Type	Description
⊞ Inst	PS1	Instrument	Message based+
⊞ abc Command		String	Command to send+
⊞ 1.0 Voltage	0.0	Real	PS voltage
⊞ 1.0 Current	0.0	Real	PS current

2. Return to the list of parameters for the original action you inserted into a new test. Select the Value field for the parameter named “Command” and copy its contents to the Windows clipboard (Edit | Copy or Ctrl-C).

Continuing with the example of a power supply, you would copy ":VOLT 5.0;:CURR 2.0" to the clipboard.

3. In the Action Definition Editor, paste the copied command into the Value field for the parameter named "Command" (Edit | Paste or Ctrl-V).

4. Close the testplan that contains the original action.

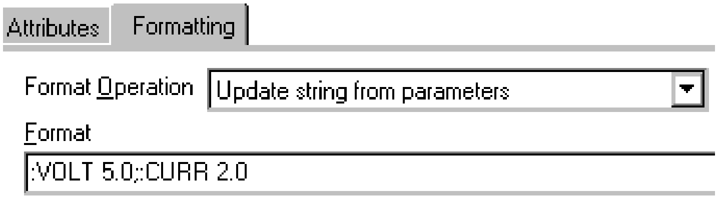
5. In the Action Definition Editor, choose the Value field for the parameter named Command.

6. Click in the Value field and choose the ... button.

7. When the String Editor appears, choose its Formatting tab

8. Change the Format Operation to "Update string from parameters." Notice that the command is copied to the Format field.

   For the example of a power supply, the result would look like this:



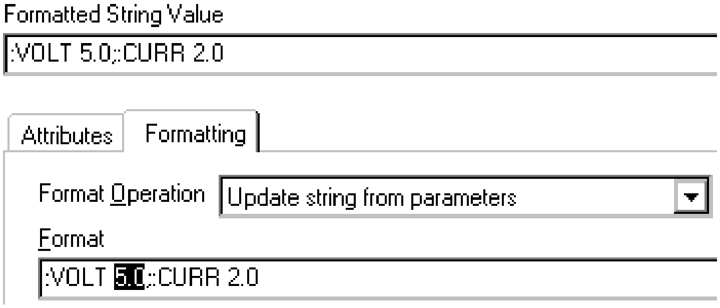
Format Operation: Update string from parameters

Format: :VOLT 5.0;:CURR 2.0

9. Do the following for each text item for which you wish to substitute a replaceable parameter in the format:

   a. Use the mouse to select the text item.

For the example of a power supply, you might select “5.0”, as shown below.



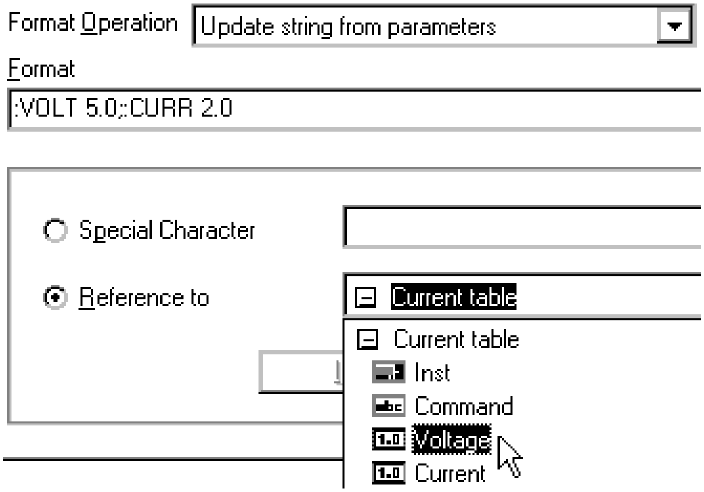
Formatted String Value: :VOLT 5.0;:CURR 2.0

Format Operation: Update string from parameters

Format: :VOLT 5.0;:CURR 2.0

b. Choose the “Reference to” option and use its drop-down list to specify which item to reference in the symbol table named `Current table`.[1]

For the example of a power supply, you would choose the symbol named Voltage in `Current table` to replace the text item “5.0”, as shown below.



Format Operation: Update string from parameters

Format: :VOLT 5.0;:CURR 2.0

○ Special Character

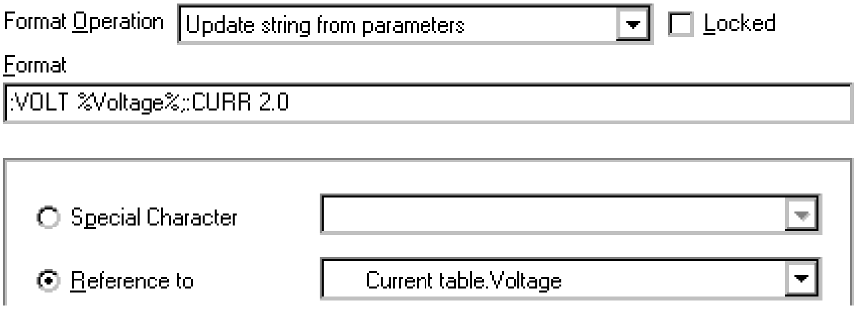
● Reference to: Current table

[1] `Current table` means the symbol table in which the string being formatted resides. Here, it is the symbol table that contains the action’s parameter block, which contains the new parameters you created.

c. Choose the Insert into Format button.

d. Verify that the format now has the name of the item you chose inserted in place of the text item you previously selected.

For the example of a power supply shown below, notice that the text entry “5.0” has been replaced by a replaceable parameter named Voltage. Also notice that “Reference to” references “Current table.Voltage”; i.e., a symbol named Voltage in the current symbol table.



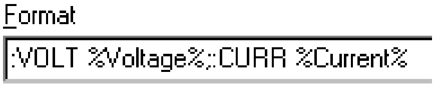
Format Operation: Update string from parameters □ Locked

Format: :VOLT %Voltage%;:CURR 2.0

○ Special Character

● Reference to: Current table.Voltage

The format for a complete string to program the voltage and current for the power supply example might look like this:

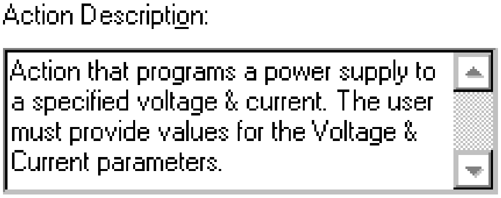


Format: :VOLT %Voltage%;:CURR %Current%

10. Choose the OK button.

11. Update the action description in the Action Definition Editor to reflect what the action now does, including how to use the task specific parameters.

A description of the example of a power supply might look like this:



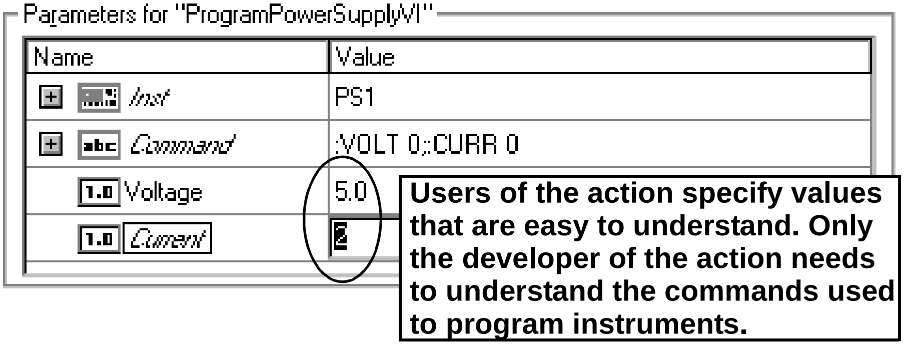
Action Description:

Action that programs a power supply to a specified voltage & current. The user must provide values for the Voltage & Current parameters.

12. Use File | Save to save the modified action definition.

Why Did You Customize the Action Definition?

Until you have done the preceding procedure a few times and used the results, it may not be obvious what you have done or why you would want to do it. Refer to the diagram below.



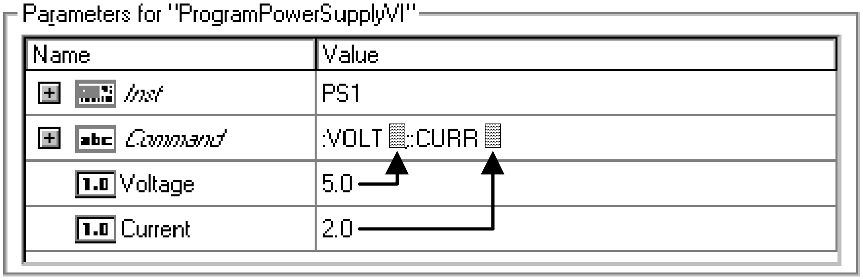
Parameters for "ProgramPowerSupplyVI"

Name	Value
Inst	PS1
Command	:VOLT 0;:CURR 0
Voltage	5.0
Current	2

Users of the action specify values that are easy to understand. Only the developer of the action needs to understand the commands used to program instruments.

As stated, *when you use the predefined actions and formatted strings to control message-based instruments, only the developer of an action needs to understand the syntax of the commands used to program the instruments*. Instead of specifying cryptic syntax, users of the action simply fill in values for parameters that have obvious names such as Voltage and Current.

In the example above, programming Voltage to 5 and Current to 2 is as simple as specifying the values for a couple of parameters. When the action executes, the values of the parameters named Voltage and Current will be substituted into the command syntax, as shown below, and the resultant command will be sent to the instrument.



Parameters for "ProgramPowerSupplyVI"

Name	Value
⊞ *Inst*	PS1
⊞ *Command*	:VOLT ▨;:CURR ▨
Voltage	5.0
Current	2.0

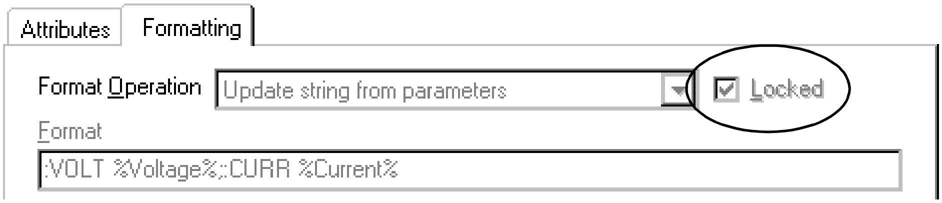
Using the Action in a Testplan

1. Create a new testplan, add a test to it, and insert the new action into that test. Run the test with several settings for its task-specific parameters to verify that the action is working correctly. If problems arise, you can modify the action's format string in the testplan until the action works as desired.

   For the action used to program a power supply, you would set Voltage and Current to various values and and verify that the output from the power supply is correct.

2. After you have verified that the action works correctly, open its definition in the Action Definition Editor and copy any changes made to the format string back to the action definition.

3. If you wish to ensure that users of the action cannot modify its format string, choose the Locked option, as shown below.



Format Operation: Update string from parameters

☑ Locked

Format: :VOLT %Voltage%;:CURR %Current%

4. If you modified the action, save it.

What if the Instrument Returns a Response?

The preceding topics described how to use actions to send commands to an instrument. But what if the instrument returns a response string instead? The process for creating an action to control an instrument that returns a response is similar to that for sending a command to an instrument, but the details vary slightly, especially when customizing the action definition. Several sections from the previous procedure reappear below with descriptions of what is different when receiving a response from an instrument.

Note

If the instrument returns a single value in its response, it may be adequate to use that as the result from the action. If the instruments returns multiple values in its response, the following topics describe how to extract each response to a separate output parameter in the action. If desired, you can use the `msginstQuery` or `msginstReceive` actions to see a sample of a valid response.

Choosing Which Action to Use

Consider the following when choosing which action to use with an instrument that returns a response.

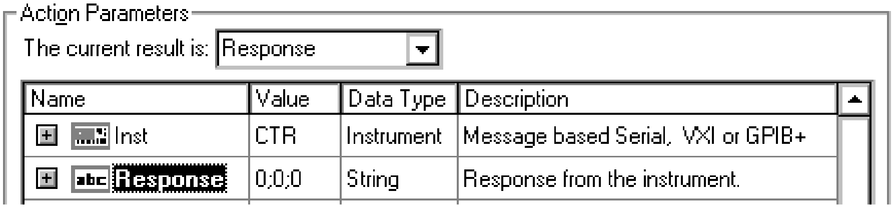
If the instrument...	Then use...
Immediately returns a response to a command	`msginstQuery`[a]
Takes awhile to return a response and you wish to do another task while waiting	`msginstSend` and `msginstReceive` with one or more actions between them to do additional tasks

a. Using `msginstQuery` is generally the safer of the two approaches because it avoids the possibiity of trying to receive a response from an instrument without first requesting that response.

Setting Up the Action

Specify a response string in the Response parameter. Run the action to verify that it receives a response from the instrument. To do this, you may need to precede it with an action that sends a command to the instrument. Or, you can use `msginstQuery`, which combines the command and response in a single action.

In the case of an action that controls a frequency counter, you might specify the parameter named `Inst` to reference instrument “CTR” and the parameter named “Response” to accept three values in the format “0;0;0”. These values will contain the peak-to-peak reading returned by the counter and the minimum and maximum voltage values of the waveform.



Action Parameters

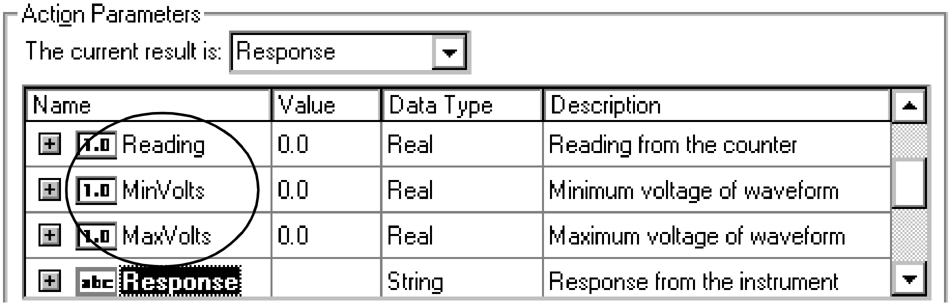
The current result is: Response

Name	Value	Data Type	Description
Inst	CTR	Instrument	Message based Serial, VXI or GPIB+
Response	0;0;0	String	Response from the instrument.

Customizing the Action Definition

1. Use the Add button in the Action Definition Editor to add task-specific parameters to the copy of the action definition.

   In the case of a frequency counter, you might add three new parameters—Reading, MinVolts, and MaxVolts—to the action definition, as shown below. These parameters would contain the results extracted from the response string via HP TestExec SL’s string formatting feature.

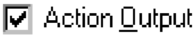


Action Parameters

The current result is: Response

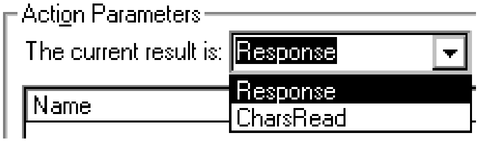
Name	Value	Data Type	Description
Reading	0.0	Real	Reading from the counter
MinVolts	0.0	Real	Minimum voltage of waveform
MaxVolts	0.0	Real	Maximum voltage of waveform
Response		String	Response from the instrument

2. Select a parameter, choose the Edit button, and specify the Action Output option for each of the new parameters you just added, as shown below.



☑ Action Output

3. If you wish to choose one of these parameters as the result for limits checking, choose it in the drop-down list that appears to the right of “The current result is:”, as shown below.



Action Parameters

The current result is: Response

4. Return to the list of parameters for the original action you inserted into a new test. Select the Value field for the parameter named “Response” and copy its contents to the Windows clipboard (Edit | Copy or Ctrl-C).

   Continuing with the example of a frequency counter, you would copy “0;0;0” to the clipboard.

5. In the Action Definition Editor, paste the copied response into the Value field for the parameter named “Response” (Edit | Paste or Ctrl-V).

6. Close the testplan that contains the original action.

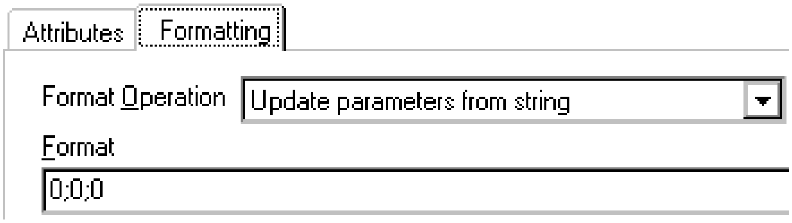
7. In the Action Definition Editor, choose the Value field for the parameter named Response.

8. Click in the Value field and choose the ... button.

9. When the String Editor appears, choose its Formatting tab

10. Change the Format Operation to “Update parameters from string.” Notice that the value of the response is copied to the Format field.

    For the example of a frequency counter, the result would look like this:



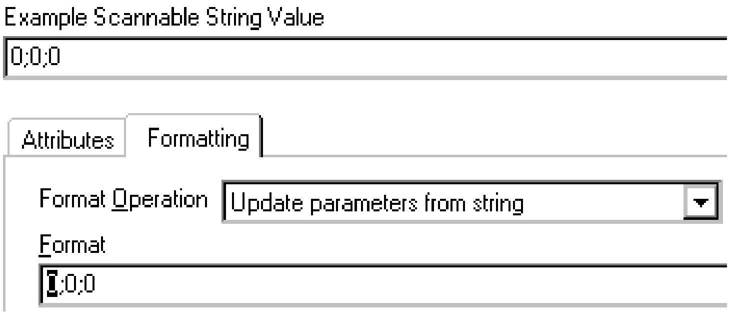
Format Operation: Update parameters from string

Format: 0;0;0

11. Do the following for each text item for which you wish to substitute a replaceable parameter in the format:

    a. Use the mouse to select the text item.

       For the example of a frequency counter, you might select the first “0”, as shown below.



Example Scannable String Value

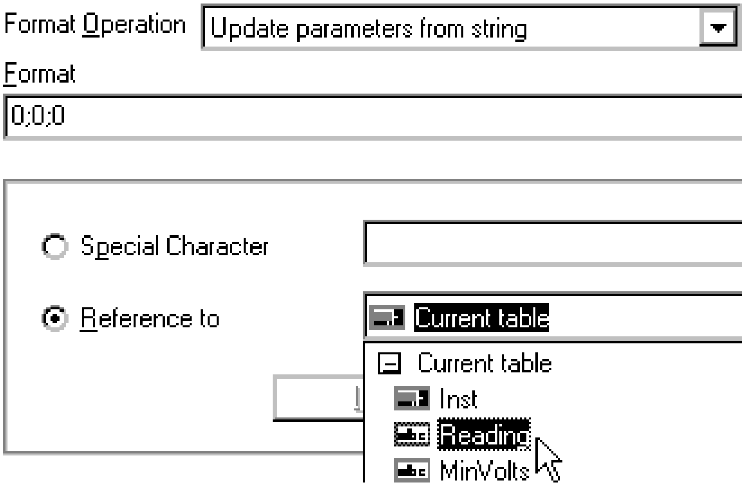
0;0;0

Format Operation: Update parameters from string

Format: 0;0;0

    b. Choose the “Reference to” option and use its drop-down list to specify which item to reference in the symbol table named `Current table`.

       For the example of a frequency counter, you could choose the symbol named Reading in `Current table` to replace the first text item, “0”, as shown below.



Format Operation: Update parameters from string

Format: 0;0;0

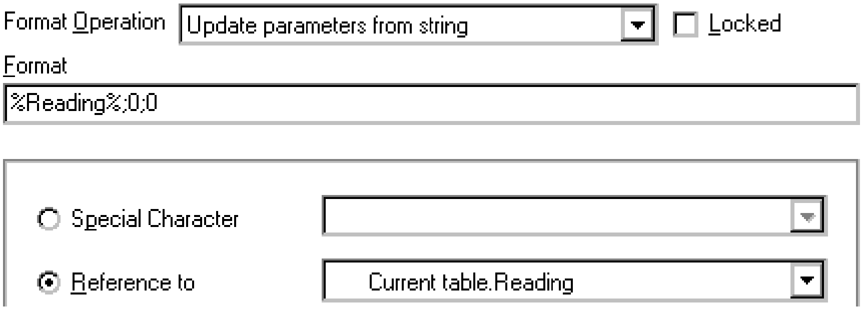
○ Special Character

● Reference to: Current table

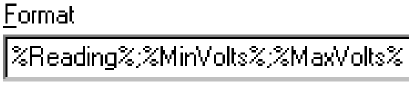
c. Choose the Insert into Format button.

d. Verify that the format now has the name of the item you chose inserted in place of the text item you previously selected.

For the example of a frequency counter shown below, notice that the text entry “0” has been replaced by a replaceable parameter named Reading. Also notice that “Reference to” references “Current table.Reading”; i.e., a symbol named Reading in the current symbol table.


Format Operation: Update parameters from string
□ Locked
Format: %Reading%;0;0
○ Special Character
● Reference to: Current table.Reading

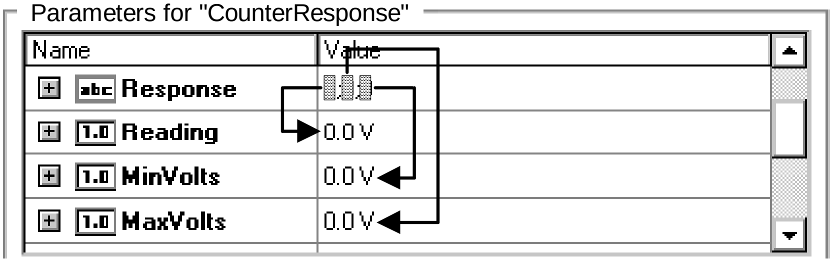
The format for a complete string to extract the values of Reading, MinVolts, and MaxVolts for the frequency counter example might look like this:


Format: %Reading%;%MinVolts%;%MaxVolts%

The example follows the standard for SCPI programming, which states that response messages may contain both commas and semicolons as separators. When a single query command returns multiple values, a comma is used to separate data items. When multiple queries are sent in the same message, the groups of data items corresponding to each query are separated by a semicolon.

Here, the data items are separated by semicolons because it is assumed that these are the responses to multiple queries. Three returned values—%Reading%, %MinVolts%, and %MaxVolts%—are referenced to parameters with similar names in the action: Reading, MinVolts, and

MaxVolts. When the actions executes, the response is returned in the Response string and the values of the parameters named Reading, MinVolts, and MaxVolts are updated, as shown below.[1] Users of the action can use those values as desired.



Parameters for "CounterResponse"

Name	Value
⊞ abc Response	▨.▨.▨
⊞ 1.0 Reading	0.0 V
⊞ 1.0 MinVolts	0.0 V
⊞ 1.0 MaxVolts	0.0 V

12. Choose the OK button.

13. Update the action description to reflect what the action now does, including how to use the task-specific parameters.

14. Use File | Save to save the modified action definition.

Debugging Actions That Control Message-Based Instruments

Keep the following in mind when debugging actions used to control message-based instruments:

- HP TestExec SL's trace feature is useful when debugging the commands sent to an instrument and its responses to those commands. The instrument handler for message-based instruments sends information to the Trace window when tracing is enabled for the instrument.

  For more information about using the trace feature, see “Using the Trace Feature to Monitor I/O Operations” in Chapter 1 of the *Using HP TestExec SL* book.

[1] The values shown for Reading, MinVolts, and MaxVolts are “0.0” because the action has not executed and updated their values.

- Predefined “msginst...” actions that may be useful during debugging include `msginstReset`, `msginstErrorStatus`, `msginstGetID`, and `msginstReceive`. These are described earlier in this chapter.

- During debugging you can display the state of an instrument in HP TestExec SL’s Watch window, as described in the following section.

Notes for Advanced Users

- The predefined action named `msginstBinarySend` lets you send binary data to an instruments. This is useful for instruments that require large amounts of data to program them, and that can accept data in a binary format. If you wish to use this action, you must understand the conventions for sending arbitrary blocks of data to instruments that are IEEE-488.2 compliant.

  The format that works best with `msginstBinarySend` is “definite block length.” Data sent in this format begins with a prefix of #8NNNNNNNN, where NNNNNNNN is the number of bytes to send, padded with zeroes on the left as needed to ensure the correct number of characters. The `msginstBinarySend` action lets you specify how many elements to send from an Int32 array, how many bytes to send from each element in that array, and the order in which to send the bytes.[1] The action also provides prefix and suffix strings in which you can send character information or extra commands to the instrument.

Note

Be sure that the number of bytes to send indicated in the prefix matches the number of bytes to send indicated by the number of elements multiplied by the number of bytes per element. For example, a prefix of #800001000 means to send 1000 bytes, which could be 500 bytes per element sent in 2 elements: 500 x 2 = 1000.

- The predefined action named `msginstBinaryReceive` lets you receive binary data from instruments. The format that works best with `msginstBinaryReceive` is “definite block length.” The action can be configured to receive a fixed number of prefix characters, a fixed number of suffix characters, and arbitrary binary data between the prefix and suffix. Binary data is stored in the elements of an Int32 array, and the action’s parameters let you specify how many bytes to put in each element and the order of those bytes.

[1] The default byte order is correct for sending binary data to instruments that are IEEE-488.2 compliant.

- Loading the hardware handler named “hwhmsginst.dll” causes it to read an initialization file named “hwhmsginst.ini” located in the same directory as the “.dll” file. You can use entries in the “.ini” file to customize the default parameter values for a specific instrument. If you enter new, instrument-specific sections in the “.ini” file, users wishing to add an instrument of that type to the system configuration can easily select these default parameters. This prevents them from having to know the specifics of an instrument to use it.

  Customizing the initialization file can be useful when:

  a. The instrument is not IEEE-488.2 compliant, and uses commands that are different from the standard reset (*RST) and ID query (*IDN?) commands.

  b. You want to specify mnemonic node names for the terminals of an instrument when it is added to the switching topology. `NodeName`*`nn`* arguments added to the initialization file appear in the NodeNames parameter for the module when it is added to a topology file, and are used to add instrument node names to the topology file.

  c. You want programming information for the instrument to appear in HP TestExec SL’s Watch window. `WatchWindow`*`nn`* arguments added to the initialization file appear in the WatchList parameter for the instrument, and contain query commands to send to the instrument whenever the testplan pauses, plus a label that appears in the Watch window for a response to the query.

  File “hwhmsginst.ini” contains instructions that describe how to modify it. Load the file into a text editor, such as WordPad in its text mode, to read the instructions or to modify the file.

10

Testing Multiple UUTs

This chapter describes how to increase the throughput of your test system by using the HP Throughput Multiplier feature to test simultaneously multiple UUTs with a single set of hardware resources and a single testplan.

About Multi-UUT Testing

Note Multi-UUT testing via the HP Throughput Multiplier feature requires a separate license available from Hewlett-Packard. A license provides you with a key that you must use to enable multi-UUT testing, as described later in this chapter. Unless you enable multi-UUT testing, you cannot run or edit multi-UUT testplans.

Why Test Multiple UUTs?

The hardware in a typical test system is expensive. If your test system is testing only one UUT (unit under test) at a time, you may not be using it to its maximum. For example, you may be tying up all the resources on a complex (and expensive) test system just to test simple UUTs one at a time.

Sometimes testing a UUT requires expensive, additional hardware. For example, automotive airbag modules require a shaker table to test their impulse response, which increases the cost of the test system. Here, it is advantageous to maximize the use of resources by shaking multiple UUTs at once and capturing their responses in parallel. Also, such modules cannot be shaken too many times because their sensors are fragile, which makes it unacceptable to shake a group of them in sequence.

Another situation where the ability to test multiple UUTs can be useful is when the printed circuit boards for UUTs are manufactured in panels. It may be necessary to test UUTs while they are still in panel form to match how the they are moved on an automated assembly line or to minimize their handling time.

A common form of multiple testing is to have two UUT fixture positions and test one position while loading/unloading the other to minimize handling time. This is sometimes called “dual-well” testing. This, too, presents an opportunity to save time via multiple-UUT testing.

All of these methods of testing multiple UUTs on a single test system contribute to reducing your cost of testing per UUT. Also, testing multiple UUTs on a single system reduces the amount of floor space that might otherwise be occupied by multiple test systems.

Note The most important thing to know about multi-UUT testing is that if your test system has the resources to handle multiple UUTs, it often can test them faster together than it can test them individually.

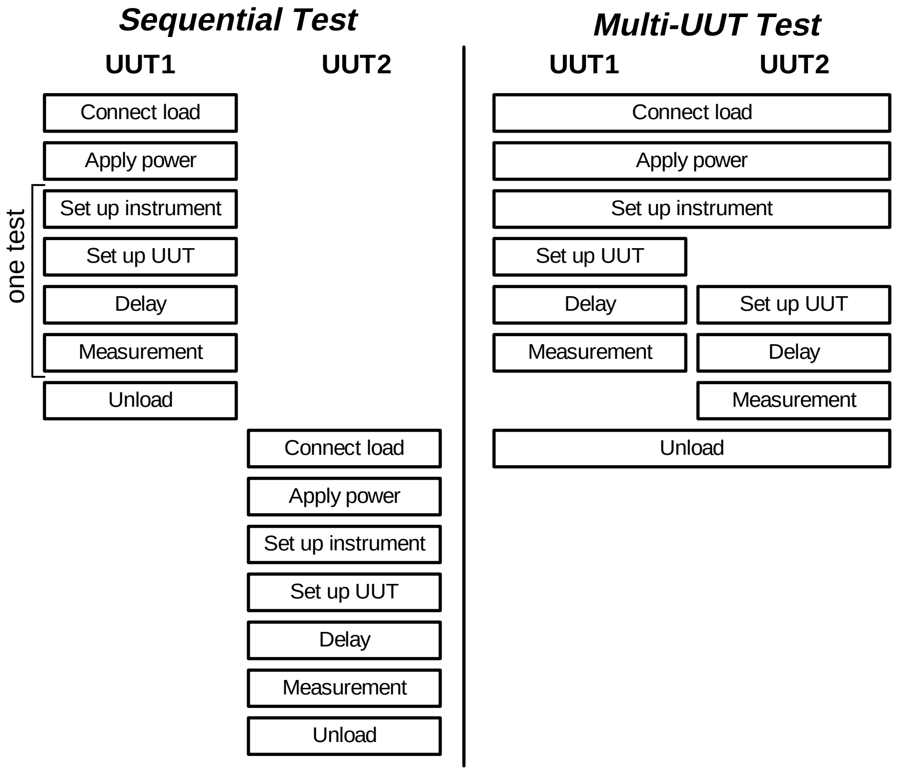
What Makes Multi-UUT Testing Faster?

At first glance, it might seem like testing multiple UUTs would be no faster than testing them one at a time. For example, typical instruments make only one measurement at a time, not multiple measurements that overlap one another. However, the time needed to test a UUT is a combination of several times:

- The time to physically handle the UUT; i.e., connecting/disconnecting it from hardware resources
- The time the UUT takes to arrive at the correct state to test it
- The time it takes to set up instruments to make a measurement
- The time it takes to make the measurement
- The time to analyze the measurement results
- The time to report the results of the measurement (including datalogging)

If any of these steps dominates the total test time and that test time is too long, there are potential opportunities to increase throughput by simultaneously doing that step for multiple modules and overlapping the steps that are slowest. Also, if there are multiple slow steps, there may be further opportunities for overlapping them.

Refer to the diagram below, which shows the flow of testing from top to bottom with respect to time for two UUTs. Tasks are done in series in conventional, sequential tests, which means the total time for testing is the sum of the times required to do the individual tasks. Multi-UUT testing, however, lets tasks that are common to all UUTs—such as connecting loads, applying power, and setting up instruments—be done in parallel, which substantially reduces the overall time for testing.



How Does HP TestExec SL Test Multiple UUTs?

A feature of HP TestExec SL called the Throughput Multiplier automatically overlaps otherwise idle periods and in many cases lets you test multiple UUTs faster than they could be tested individually. The Throughput Multiplier speeds up test times by:

- Overlapping tests for a single module by using setup “dead time” to overlap multiple tests or actions within tests
- Reducing handling time by coordinating multiple nested tests; i.e., running the test for UUT A while putting UUT B in place for testing

- Reducing testing time by testing more than one module at once, which uses setup “dead time” and duplicated resources. Examples of this include setting up instruments or power supplies only once and then using them to test multiple modules.

- Allowing for multiple spots in the testplan where operations are done simultaneously across all modules. For example, if a UUT is slow to respond to a command, that command can be simultaneously sent to multiple UUTs and the time spent awaiting a response is similar to that of a single UUT instead of multiple UUTs.

Note

The appearance of some of HP TestExec SL’s forms change depending upon whether you are testing single or multiple UUTs. For example, when testing multiple UUTs the Switching Path Editor presents an additional option that lets you specify the UUT to which the switching paths apply.

What Must You Do to Test Multiple UUTs?

You can use the Throughput Multiplier to test multiple UUTs *when all the UUTs are of a single type*; i.e., you cannot mix different types of UUTs in a testplan. The overall steps you must follow when creating a multi-UUT testplan are:

- Enable multi-UUT testing in HP TestExec SL’s initialization file

- Create a UUT topology layer for the first UUT position using UUT position names as part of the node names associated with specific UUT positions

- Create and debug a normal testplan for the first UUT position

- Convert the testplan to a multi-UUT version

- Add flow control statements as needed

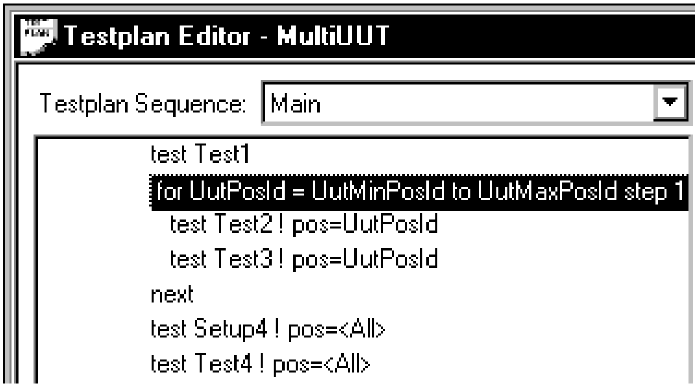
- Add UUT positions as needed

- Run the testplan and debug the new UUT positions

These steps are described in greater detail later.

Example of a Multi-UUT Testplan

The left pane of the Testplan Editor window might look like this for a multi-UUT test:



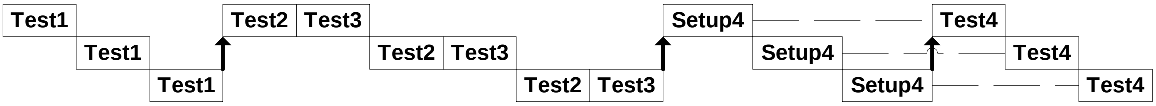
Testplan Editor - MultiUUT

Testplan Sequence: Main

```
test Test1
for UutPosId = UutMinPosId to UutMaxPosId step 1
  test Test2 ! pos=UutPosId
  test Test3 ! pos=UutPosId
next
test Setup4 ! pos=<All>
test Test4 ! pos=<All>
```

Notice that many of the tests have comments to their right that identify for which UUT positions they will execute during multi-UUT testing. These comments are automatically inserted when you specify the multi-UUT options for tests.

This testplan implements the flow of tests shown below.



The example above shows synchronous execution of homogeneous tests for three UUT positions. The rows represent the tests for a particular UUT position, such as on a multi-UUT panel, and the columns represent the flow of time. `Test1` is executed for each position and followed by `Test2` and `Test3` for each position. Finally, a setup test and its associated test—`Setup4` and `Test4`, respectively—are executed for each position. The dashed lines show the relationships between the setup tests and their associated tests that do tasks such as make measurements.

Notice how variables `UutPosId`, `UutMinPosId`, and `UutMaxPosId` are used in the testplan to count in the *for...next...step* loop. These variables are predefined symbols that appear in the System symbol table when the HP Throughput Multiplier is enabled.

You can find samples of multi-UUT testplans in directory “<*HP TestExec SL home*>\samples\multipleUUT” along with a “readme.txt” file that describes how to use them.

Symbols Used by Multi-UUT Testing

Multi-UUT testing uses the following predefined symbols in the SequenceLocals symbol table, whose scope is global to all tests. These symbols appear in multi-UUT testplans to control the flow of testing. In most cases, HP TestExec SL maintains the values of the symbols and you do not need to manipulate them unless your testing needs are unique.

Symbols used to control the sequence of multi-UUT testing

UutPosId The identifier of the current UUT position when testing a group of UUTs. The value of this symbol should be between `UutMinPosId` and `UutMaxPosId`.

UutMinPosId The identifier of the minimum UUT position. The value of this symbol, whose default is 1, should be the same as the lower boundary of any arrays used to store data while testing multiple UUTs. A change to this symbol’s value takes effect on the next run of the testplan.

UutMaxPosId The identifier of the maximum UUT position. The value of this symbo should be the same as the upper boundary of any arrays used to store data when testing multiple UUTs. A change to this symbol’s value takes effect on the next run of the testplan.

UutXoutFlags An array whose contents indicate whether to test each UUT position. 0 indicates that the UUT in that position should be tested, and 1 indicates that it should not be tested.

Symbols used for reporting the results from multi-UUT testing

UutSerialNumbers An array of the serial numbers for the UUTs at each UUT position.

UutTestingStates An array that contains the current state of testing for each UUT position. These states correspond to the testing states returned by the State property of the HP TestExec SL Control, which is described in Chapter 1 of the *Customizing HP TestExec SL* book.

UutPosNames An array of the names for each UUT position in the fixture.

More About UutPosId

The previous topic lists `UutPosId` among the symbols used by multi-UUT testing. Because `UutPosId` is integral to multi-UUT testing, additional things you may need to know about it are listed below.

- The value of `UutPosId` is controlled by the sequencer inside tests
- The various options you can specify for individual tests during multi-UUT testing (described in greater detail later) affect it as follows:

The option...	Causes the sequencer to...
Repeat this test for all UUT Positions	Repeat the test with successive values of `UutPosId`. When done, it resets `UutPosId` to its starting value.
Execute this test for the UUT Position defined by ‘UutPosId’	Honor the value of `UutPosId` at the beginning of the test; i.e., it does not modify the value of `UutPosId`
Execute this test once independent of UUT positions	Set the value of `UutPosId` to `UutMinPosId` - 1 while the test runs and restore the previous value when the test ends

Multi-UUT Effects on Datalogging

HP TestExec SL maintains a separate buffer that contains datalogging information for each UUT position. This lets HP TestExec SL create a separate datalogging file for each UUT position. If you customize the format of the log records to log the values of any of the symbols used for multi-UUT testing, as described above, HP TestExec SL will ensure that symbol `UutPosId` always reports the position for which the log record is being produced. This lets you parse the log records to determine which UUT position the log data describes. If a particular UUT position is omitted from testing because it is disabled, no datalogging file will be created for that UUT position.

Multi-UUT Effects on Reporting

Status information sent to HP TestExec SL's Report window is much the same for single or multi-UUT testing. However, at the end of each run of the testplan during multi-UUT testing HP TestExec SL will loop through the status reporting for each UUT position. The result from each test will indicate which position the result is for. It also will indicate as "Testplan Stopped" any UUT positions that were not tested. Finally, it will indicate any UUT positions that were not completely tested because of branches in the flow of the testplan, such as *On Fail Branch To* branching.

Multi-UUT Effects on Testplan Listings

Multi-UUT testing displays extra information in listings of testplans and their contents, such as:

- An indication that multi-UUT testing was enabled
- An indication of which tests will execute for all UUT positions
- The parameters used by switching actions for each UUT position

Multi-UUT Effects on Breakpoints & Single-Stepping

If a breakpoint is set on a test that has *Execute for all UUT positions* specified for it (on the Options tab in the right pane of the Testplan Editor window), the break will occur for every UUT position before executing the test. This maintains the `UutPosId` setting across pauses in the testplan, and lets looping resume at the current count when the testplan continues.

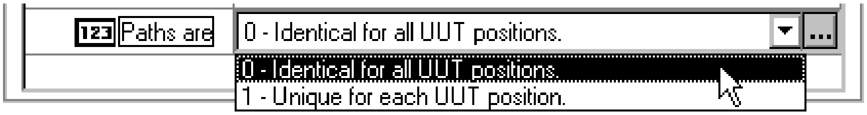
Single-stepping a multi-UUT testplan will also pause the testplan after each iteration in the loop for all UUT positions.

When paused at a breakpoint or during single-stepping, you can examine HP TestExec SL’s status line (at the bottom of the main window) to see the name of the UUT position that will execute next.

Multi-UUT Effects on Switching

Switching actions in tests let you control switching hardware, such as relay matrixes, to set up connections needed during testing. For example, a switching action might connect the UUT to a power supply, an instrument to provide a stimulus, and an instrument to measure a response.

When testing multiple UUTs, connections used to test one UUT position often need to be reused, with only minor variations that move the connections from one UUT position to the next, to test additional UUT positions. To support this need, HP TestExec SL provides a variation on switching actions for multi-UUT testing. As shown below, you can specify whether the paths are identical for all UUT positions or unique for each position.

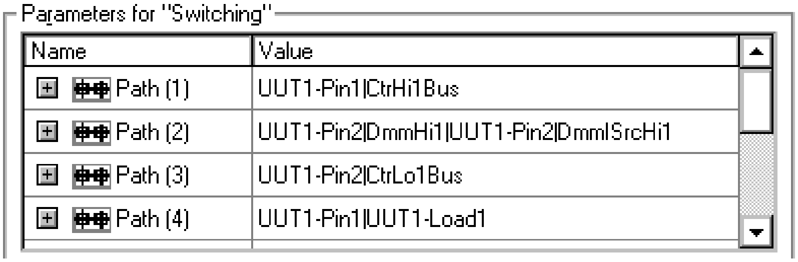


123 Paths are 0 - Identical for all UUT positions.

0 - Identical for all UUT positions.
1 - Unique for each UUT position.
...

Having identical paths means the same path is used for all the UUT positions tested by a given test, while having unique paths means you can specify a separate path for each UUT position.

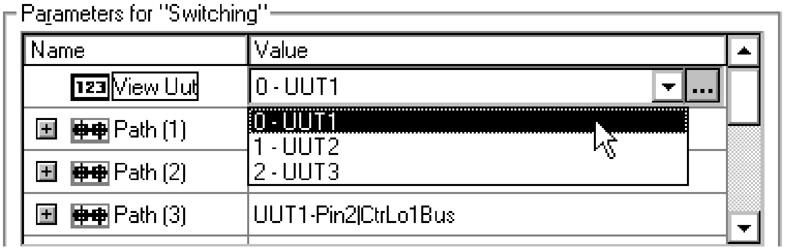
Refer to the example below, which shows a switching action when identical switching paths are used in a test for multiple UUT positions.



Parameters for "Switching"

Name	Value
⊞ Path (1)	UUT1-Pin1|CtrHi1Bus
⊞ Path (2)	UUT1-Pin2|DmmHi1|UUT1-Pin2|DmmISrcHi1
⊞ Path (3)	UUT1-Pin2|CtrLo1Bus
⊞ Path (4)	UUT1-Pin1|UUT1-Load1

Now refer to the following example, which shows a switching action when unique switching paths are used, one per UUT position. Notice how a drop-down list appears that lets you choose which set of paths to view or edit.



Parameters for "Switching"

Name	Value
View Uut	0 - UUT1 0 - UUT1 1 - UUT2 2 - UUT3
⊞ Path (1)	
⊞ Path (2)	
⊞ Path (3)	UUT1-Pin2|CtrLo1Bus

Testing Multiple UUTs

The best way to create a multi-UUT testplan is to begin with a testplan that tests a single UUT position and then expand it by adding UUT positions and verifying their operation as you add them.

Enabling Multi-UUT Testing

If the HP Throughput Multiplier feature is not already enabled, do the following to enable multi-UUT testing on your test system:[1]

1. Use a text editor, such as WordPad in its text mode, to load file "<*HP TestExec SL home*>\bin\tstexcsl.ini".

2. Edit the `[Process]` section of the initialization file to read:

   ```
   [Process]
   Throughput Multiplier License=<your license number>
   ```

   For example, if the number on the paper licence you received for the Throughput Multiplier option was `XYZ123`, you would enter:

   ```
   [Process]
   Throughput Multiplier License=XYZ123
   ```

3. Save the modified file and exit the editor.

Note

If you edit the initialization file while HP TestExec SL is running, you must exit and restart HP TestExec SL for the change to be seen.

[1] You can tell if the feature is enabled by loading a testplan and choosing Options | Testplan Options in the menu bar. If the Testplan Options box has a tab labeled Throughput Multiplier, then the feature is enabled.

Creating the First UUT Position

Creating a UUT Topology Layer for the First UUT

Note Your goal in creating a topology layer for the first UUT position is to define topology that is readily expandable for additional UUT positions.

Create a UUT topology layer for the first UUT position. Specify UUT node names that combine a generic UUT name with a specific UUT position, such as `UUT1_Gnd`. This makes it easy to add more UUT positions later, such as UUT2_Gnd and UUT3_Gnd. Other examples of appropriate names for nodes are:

For UUT1: UUT1_Pin1, UUT1_Load3

For UUT2: UUT2_Pin1, UUT2_Load3

For UUT3: UUT3_Pin1, UUT3_Load3

Note Notice how, with the exception of its unique identifier, each UUT’s list of nodes is identical to those of the other UUTs. The UUT names must be identical, including capitalization, except for the number that identifies the UUT position.

For more information about creating topology layers, see Chapter 4.

Creating & Debugging a Testplan for the First UUT Position

Create a testplan and debug it as you would for a single-UUT testplan. Because you begin by working with only a single UUT position, for all practical purposes the initial testplan is a single-UUT testplan.

For more information about creating and debugging testplans, see Chapter 1.

Converting the Testplan to a Multi-UUT Version

After you have created a basic testplan for a single UUT position, you must convert the testplan to a multi-UUT version.

Caution Be sure you want to convert the testplan to a multi-UUT version. Once you have converted a testplan to a multi-UUT version, you cannot easily convert it back to a single-UUT version.

Globally Enabling Multi-UUT Testing

1. Choose Options | Testplan Options in HP TestExec SL’s menu bar.

2. Choose the Throughput Multiplier tab.

3. Choose the *Add Multiple UUT Features* button.

4. In the field to the right of *Current number of positions*, specify how many UUT positions you wish to test. For example, if you are creating a test for a multi-UUT panel that contains 4 UUTs, specify “4” here.

5. Choose the *Update Position Names* button.

6. Edit the default names of the UUT positions as needed by clicking them in the list under *Enable UUT Position(s)* and choosing the *Edit UUT Name* button to invoke a dialog box that lets you edit their names.

Note Be sure the names of UUT positions that you specify here match those you used when defining the switching topology; e.g., if your topology has nodes named UUT1_Gnd and UUT1_Load1, you need a matching UUT position named UUT1.

Tip: You can click the name of a UUT position in the list and then use the *Move Name Up* or *Move Name Down* button to reposition it.

7. Uncheck the names of all UUT positions but the first one to disable them for now.

8. Choose the OK button.

9. A dialog box appears to inform you that multi-UUT testing has been enabled. It also asks if you wish to use the recommended default settings for multi-UUT testing. Follow the instructions and choose the Yes or No button as appropriate.

Controlling the Flow of Testing

Adding Flow Control Statements

If portions of your testplan require looping or branching based on UUT positions, add the appropriate flow control statements. For example, you might surround a series of tests like this:

```
for UutPosId = UutMinPosId to UutMaxPosId step 1
  test Test5
  test Test6
next
```

to have them repeated for all UUT positions. You can evaluate the symbols described earlier under “Symbols Used by Multi-UUT Testing” in flow control statements that control the flow of testing.

Adding flow control statements can be an iterative process, and you may need to add additional statements as you add new UUT positions.

Specifying Multi-UUT Options for Individual Tests

If desired, you can specify how each test executes with respect to UUT positions during multi-UUT testing. Do the following for each test:

1. Select the test in the left pane of the Testplan Editor window.

2. Choose the Options tab in the right pane of the Testplan Editor window.

3. Choose the desired option under *Throughput Multiplier configuration*, as described below.

Choose this option...	To have the test execute...
Repeat this test for all UUT Positions	For all possible UUT positions unless:[a] • A particular UUT position has exceeded the *Halt on failure count* limit for the testplan (as set on the Execution tab in the Testplan Options box), in which case the test will not execute for that UUT position. • The UutXoutFlags setting for a particular UUT position is non-zero (Xout or disable this UUT position), in which case the test will not execute for that UUT position.
Execute this test for the UUT Position defined by ‘UutPosId’	Only for the UUT position specified by the current value of symbol `UutPosId` in the `SequenceLocals` symbol table.
Execute this test once independent of UUT positions	Regardless of the UUT position. Test results will affect the testing results for all UUT positions. Test results will be added to the report information acquired during testing. Datalogging files will be written for all UUT positions.

[a] The range of possible UUT positions is specified by the values of symbols *UutMinPosId* and *UutMaxPosId* in the System symbol table, which are read when the testplan begins running. Changes to the values of these symbols while a testplan is running will be ignored.

Note

Given the options above, all tests whose execution depends on the UUT position must have their option set to “Repeat this test for all UUT positions” or “Execute this test for the UUT Position defined by ‘UutPosId’.” Otherwise, they will execute only once for the first UUT position, as if running a single-UUT testplan.

Adding UUT Positions to the Testplan

After you have created and debugged the first UUT position and converted the testplan to a multi-UUT version, do the following steps for each additional UUT position.

Note

Instead of adding many new UUT positions at once, we suggest that you add new positions one at a time, and then run the testplan to verify that they work correctly. This methodical approach makes it easier to troubleshoot any problems that may occur.

Adding a New UUT Position

1. Modify the switching topology by adding node names for each new UUT position. For each position, specify the generic name of each UUT position plus its unique prefix aliased to the correct system resources.

2. Choose Options | Testplan Options in the menu bar.

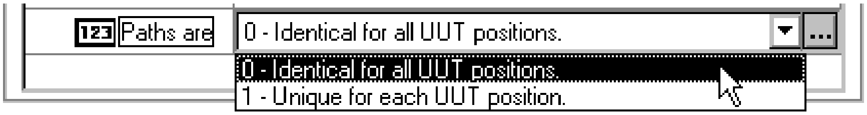
3. Choose the Throughput Multiplier tab.

4. Enable the check box for the next available UUT position in the list under “Enable - UUT Position(s)”

5. Choose the OK button.

6. Specify the switching paths for any switching actions used by the newly created UUT position.

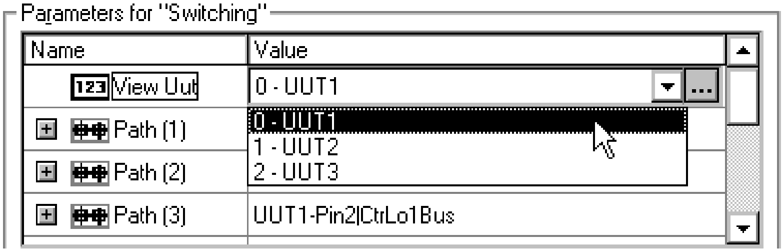
   You have the option of specifying whether the paths are identical for all UUT positions or unique for each position. As shown below, this is controlled by an option on the Actions tab in the right pane of the Testplan Editor window.



123 Paths are 0 - Identical for all UUT positions.

0 - Identical for all UUT positions.
1 - Unique for each UUT position.

If you choose the “Unique for each UUT position” option, which lets you specify an individual set of paths for each UUT position, you also must specify which position you are working with, as shown below.



Parameters for "Switching"

Name	Value
View Uut	0 - UUT1
Path (1)	
Path (2)	
Path (3)	UUT1-Pin2|CtrLo1Bus

After you have done this, you can specify the paths as usual.

Note

As you add UUT positions when using the “Unique for each UUT position” option, HP TestExec SL automatically creates the appropriate set of new paths. In many cases, you probably will not need to edit the new paths because they will be correct as-is.

For more information about specifying switching paths, see “Controlling Switching During a Test/Test Group“ in Chapter 2.

Running the Testplan & Debugging the New UUT Position

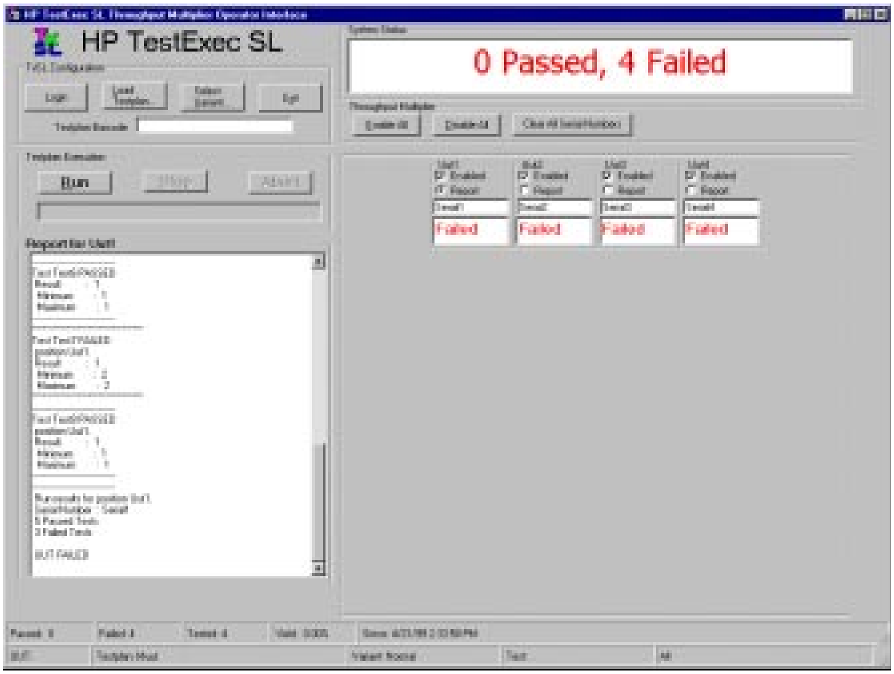
Run the testplan and debug any problems associated with the new UUT position.

The Multi-UUT Operator Interface

Note

This section assumes you are familiar with the single-UUT operator interface described in Chapter 1 of the *Customizing HP TestExec SL* book.

Besides the sample operator interface written in Visual Basic that supports single-UUT testing, there also is a sample operator interface that supports multi-UUT testing via the HP Throughput Multiplier feature of HP TestExec SL. As shown below, the multi-UUT operator interface uses a framed set of identical controls to convey information about individual UUTs. Each set of controls shows the UUT's pass/fail status and serial number plus lets the user enable/disable the UUT for testing and report generation.



You can find the code for the sample operator interface's project ìn directory "<*HP TestExec SL home*>\samples\visualbasic\operatorinterfaces\multiple". A compiled, executable form of the operator interface resides at "<*HP TestExec SL home*>\bin\MultipleOpUi.exe". Also, you can run the multi-UUT operator interface from the “HP TestExec SL” program group in the Windows Start menu.

Compatibility of Single- & Multi-UUT Operator Interfaces

Running a Multi-UUT Testplan on a Single-UUT Interface

If you run a multi-UUT-enabled testplan on a single-UUT operator interface (such as one derived from the “typicalopui” sample provided with HP TestExec SL), the testplan will run to completion. However, none of the data for individual UUTs will be visible, nor will there be any control over the serial numbers of UUTs or the `UutXOut` flags that determine which UUTs are tested. The Report window will show an intermingled collection of the streams of report information from individual UUTs.

Running a Single-UUT Testplan on a Multi-UUT Interface

If you run testplan that is not multi-UUT-enabled on a multi-UUT operator interface, the information for only one UUT will be displayed. The Report window will show the report information for that UUT.

Some Differences Between the Modules

In many respects the features of the multi-UUT operator interface are similar to those of the single-UUT operator interface, including the naming of forms and modules. The main differences are described below.

Form/Module	Comments
frmMain	Includes additional code to handle multiple UUTs
modAppSpecific	Changes to set up the captions for additional controls used by multi-UUT testing
modConfiguration	The feature that allowed automatically running a testplan after loading the bar code for a testplan has been disabled

Unique Features of the Multi-UUT Operator Interface

Variable mbMultiUutTestplan

A Boolean named `mbMultiUutTestplan`, which resides in form `frmMain`, represents the value of the `MultiUutTestplan` property read from the HP TestExec SL Control. If `mbMultiUutTestplan` is true, the operator interface manages multiple UUTs; if it is false, it does not.

Shortcuts When Accessing Symbols in Symbol Tables

The multi-UUT operator interface makes use of the special symbols associated with multi-UUT testing (described earlier under “Symbols Used by Multi-UUT Testing”). To make it easier to access those symbols, code in form `frmMain` in the multi-UUT operator interface provides “wrapper” functions for returning, and sometimes setting, the values of the symbols. The wrapper functions are:

The function named ...	Is used to access the symbol...
GetUutPosId()	UutPosId
GetUutMinPosId()	UutMinPosId
GetUutMaxPosId()	UutMaxPosId

GetUutSerialNumbers()
SetUutSerialNumber()
SetUutSerialNumbers()
	UutSerialNumbers

GetUutPosNames()	UutPosNames
GetUutTestingStates()	UutTestingStates

GetUutXoutFlags()
SetUutXoutFlag()
SetUutXoutFlags()
	UutXoutFlags

Potential Differences in the Indexing of Arrays

To increase speed, the multi-UUT operator interface keeps local copies of the `UutPosNames` and `UutXoutFlags` symbols, both of which contain arrays, found in HP TestExec SL’s `SequenceLocals` symbol table.

Because the lower index of arrays in symbol tables can vary but the control arrays used in the multi-UUT operator interface have a fixed lower index of 1, converting between the two indexes may be necessary.

Code in HP TestExec SL references arrays associated with multi-UUT testing in symbol tables via a variable named `UutPosId`, whereas code in the multi-UUT operator interface references controls in control arrays via a variable named `UutDisplayPosId`. The following functions are provided in form `frmMain` to convert between the two kinds of indexes for these arrays.

UutDisplayPosId()	Converts a specified `UutPosId` to its equivalent `UutDisplayPosId` value
UutPosId()	Converts a specified `UutDisplayPosId` to its equivalent `UutPosId` value

Changing the Number of UUT Positions

The sample multi-UUT operator interface has a predefined number of UUT positions. But what if you wish to change the number of positions? A function named `CreateUutDisplayControlArray` in form `frmMain` is used to define the control array that contains the set of controls for each UUT position. If you wish to change the number of UUT positions, you can modify the way this function operates.

Note The physical layout of the sample multi-UUT operator interface limits the maximum number of UUT positions to about 30. If you increase the number beyond this, your UUT position indicators (sets of controls) may become too small to be easily readable. If so, you may wish to adopt a different strategy for organizing the indicators, such as a tabbed layout or MDI forms.

Considerations for Factory Automation

How are Serial Numbers Read?

The multi-UUT operator interface operates such that any valid entry into the serial number field results in a UUT position being enabled. This lets users quickly scan bar codes across a group of UUTs without using the keyboard.

What if the Testplan Reads Serial Numbers from UUTs?

There are cases where a testplan reads the serial numbers of UUTs directly from the UUTs themselves and needs to update the operator interface with those serial numbers. If so, you can have a `SendUserDefinedMessage` action (in “<*HP TestExec SL home*>\samples\uidebug\actiondefinitions”) in your testplan send a user-defined message named `ReadUutSerialNumbersFromTestplan` whose ID is 50101. When the operator interface receives this message, it will reread the value of the `UutSerialNumbers` symbol for the testplan and update the operator interface as needed.

For more information about user-defined messages, see “Understanding User-Defined Messages” in Chapter 1 of the *Customizing HP TestExec SL* book.

What if the Testplan Gets the Testing Status from UUTs?

Suppose the testplan attempts to initialize a UUT and finds that it cannot talk to it. In that case, the testplan should not test the uninitialized UUT but should test other UUTs that it was able to initialize. The testing status indicator—i.e., “flag”—in array `UutXOutFlags` (in the SequenceLocals symbol table) for the unresponsive UUT should be set to 1 (no test) and the multi-UUT operator interface should be updated to disable the testing of the defective unit.

Whenever the multi-UUT operator interface needs to be updated with the status of the `UutXoutFlags` array for the group of UUTs being tested, you can have a `SendUserDefinedMessage` action (in “<*HP TestExec SL home*>\samples\uidebug\actiondefinitions”) in your testplan send a user-defined message named `ReadUutXOutFlagsFromTestplan` whose ID is 50102 .

For more information about user-defined messages, see “Understanding User-Defined Messages” in Chapter 1 of the *Customizing HP TestExec SL* book.

Index