Rational® Testing Products

Command Line Interface To Rational Test Script Services

VERSION: 2003.06.00

PART NUMBER: 800-026177-000

WINDOWS/UNIX

Rational®
the software development company

support@rational.com
http://www.rational.com

Legal Notices

Part Number: 800-026177-000
Version Number: 2003.06.00

Third Party Notices, Code, Licenses, and Acknowledgements

Contents

Preface

About This Manual

This manual is a reference of the commands that you use to add a variety of testing services to your test scripts — services such as datapool, logging, monitoring, and synchronization.

The Test Script Services described in this manual are designed to be used with Rational® TestManager® .

Audience

This manual is intended for test designers who write or edit test scripts in a scripting language such as Perl or a UNIX shell. Your command line test scripts can be used for both performance and functional testing.

Other Resources

- To access an HTML version of this manual, click **TSS for Command Line** in the following default installation path (*ProductName* is the name of the Rational® product you installed, such as Rational TestStudio®):

  Start > Programs > Rational *ProductName* > Rational Test > API

- All manuals for this product are available online in PDF format. These manuals are on the *Rational Solutions for Windows* Online Documentation CD.
- For information about training opportunities, see the Rational University Web site: http://www.rational.com/university.

Integrations Between Rational Testing Tools and Other Rational Products

Rational TestManager Integrations		
Integration	**Description**	**Where it is Documented**
Rational TestManager–Rational Administrator	Use Rational Administrator to create and manage Rational projects. A Rational project stores software testing and development information. When you work with TestManager, the information you create is stored in Rational projects. When you associate a RequisitePro project with a Rational project using the Administrator, the RequisitePro requirements appear automatically in the Test Inputs window of TestManager.	▪ *Rational Suite Administrator's Guide* ▪ *Rational TestManager User's Guide* ▪ Rational TestManager Help
TestManager–Rational ClearQuest	Use ClearQuest with TestManager to track and manage defects and change requests throughout the development process. With TestManager, you can submit defects directly from a test log in ClearQuest. TestManager automatically fills in some of the fields in the ClearQuest defect form with information from the test log and automatically records the defect ID from ClearQuest in the test log.	▪ *Rational TestManager User's Guide* ▪ Rational TestManager Help
TestManager–Rational Rational Unified Change Management (UCM)	Use UCM with TestManager to: ▪ Archive test artifacts such as test cases, test scripts, test suites, and test plans. ▪ Maintain an auditable and repeatable history of your test assets. ▪ Create baselines of your test projects. ▪ Manage changes to test assets stored in the Rational Test datastore.	▪ *Rational TestManager User's Guide* ▪ Rational TestManager Help ▪ *Rational Suite Administrator's Guide* ▪ Rational Administrator Help ▪ *Using UCM with Rational Suite*

Rational TestManager Integrations		
Integration	**Description**	**Where it is Documented**
TestManager–Rational RequisitePro	Use RequisitePro to reference requirements from TestManager so that you can ensure traceability between your project requirements and test assets. Use requirements in RequisitePro as test inputs in a test plan in TestManager so that you can ensure that you are testing all the agreed-upon requirements.	▪ *Rational TestManager User's Guide* ▪ Rational TestManager Help ▪ *Rational Suite Administrators Guide*
TestManager–Rational Robot	Use TestManager with Robot to develop automated test scripts for functional testing and performance testing. Use Robot to: ▪ Perform full functional testing. Record test scripts that navigate through your application and test the state of objects through verification points. ▪ Perform full performance testing. Record test scripts that help you determine whether a system is performing within user-defined response-time standards under varying workloads. ▪ Test applications developed with IDEs (Integrated Development Environments) such as Java, HTML, Visual Basic, Oracle Forms, Delphi, and PowerBuilder. You can test objects even if they are not visible in the application's interface. ▪ Collect diagnostic information about an application during test script playback. Robot is integrated with Rational Purify, Rational Quantify, and Rational PureCoverage. You can play back test scripts under a diagnostic tool and see the results in the test log in TestManager.	▪ *Rational TestManager User's Guide* ▪ Rational TestManager Help ▪ *Rational Robot User's Guide* ▪ Rational Robot Help ▪ *Getting Started: Rational PurifyPlus, Rational Purify, Rational PureCoverage, Rational Quantify.* ▪ Rational PurifyPlus Help

Rational TestManager Integrations		
Integration	**Description**	**Where it is Documented**
TestManager–Rational Rose	Use as test inputs in TestManager. A test input can be anything that you want to test. Test inputs are defined in the planning phase of testing. You can use TestManager to create an association between a Rose model (called a test input in TestManager) and a test case. You can then create a test script to ensure that the test input is met. In TestManager, you can view the test input (the Rose model element) associated with the test case.	▪ *Rational TestManager User's Guide* ▪ Rational TestManager Help
TestManager–Rational SoDA	Use SoDA to create reports that extract information from one or more tools in Rational Suite. For example, you can use SoDA to retrieve information from different information sources, such as TestManager, to create documents or reports.	▪ *Rational SoDA User's Guide* ▪ Rational SoDA Help ▪ *Rational TestManager User's Guide*
TestManager–Rational Unified Process (RUP)	Use Extended Help to display RUP tool mentors for TestManager. RUP tool mentors provide practical guidance on how to perform specific process activities using TestManager and other Rational testing tools. Start Extended Help from the TestManager Help menu.	▪ *Rational TestManager User's Guide* ▪ Rational TestManager Help ▪ Rational Extended Help

Contacting Rational Technical Publications

To send feedback about documentation for Rational products, please send e-mail to our technical publications department at techpubs@rational.com.

Contacting Rational Customer Support

If you have questions about installing, using, or maintaining this product, contact Rational Customer Support as follows:

Your Location	Telephone	Facsimile	E-mail
North America	(800) 433-5444 (toll free) (408) 863-4000 Cupertino, CA	(781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 (0) 20-4546-200 Netherlands	+31 (0) 20-4545-201 Netherlands	support@europe.rational.com
Asia Pacific	+61-2-9419-0111 Australia	+61-2-9419-0123 Australia	support@apac.rational.com

Note: When you contact Rational Customer Support, please be prepared to supply the following information:

- Your name, telephone number, and company name
- Your computer's make and model
- Your operating system and version number
- Product release number and serial number
- Your case ID number (if you are following up on a previously reported problem)

Introduction to tsscmd 1

About tsscmd

tsscmd is a command-line executable that gives test scripts access to Rational Test Script Services (TSS). **tsscmd** can be called from a compiled program; for example, a C program can call **tsscmd** using the `system()` function. Typically, however, **tsscmd** statements appear inside a source file written in some scripting language. For example, test scripts written in the Bourne shell, Perl, Python, or Windows `cmd` languages can access test script services through internal **tsscmd** statements.

With **tsscmd**, you can access services such as logging, synchronization, timing, and datapools. The next chapter documents all the test script services provided by **tsscmd**.

Setting Up TestManager for tsscmd

A TestManager suite can contain test scripts of different types. When a TestManager user runs a suite, TestManager invokes a program (a test script execution adapter, or TSEA) that knows how to execute each type of script in the suite. One of the built-in test script types supported by TestManager is **Command Line**. Whenever a user executes a test script containing **tsscmd** statements, TestManager invokes the command line TSEA.

Although **tsscmd** can be called from a compiled program, the most likely usage is through **tsscmd** statements inside a source file written in a scripting language such as Perl. To use **tsscmd** in this way, we recommend that you add a test script type to TestManager that uses the command line TSEA.

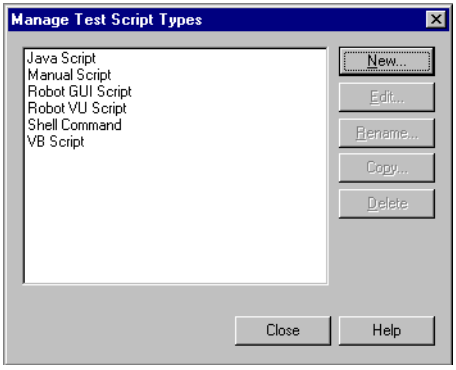
The procedure for doing this is described below. Performing this procedure enables TestManager to execute Perl scripts containing **tsscmd** statements. You can then add Perl test scripts to suites containing test scripts of other types (Java, Visual Basic, VU, GUI). And you can run, view, or edit Perl test scripts from the TestManager **File** menu.

You can run single test scripts from TestManager that can be executed from the command line (whether or not they include **tsscmd** statements) without creating a new test script type. To do this, select **File > Run Test Script > Command Line** and either type the path name of the script to run or use the browse button.

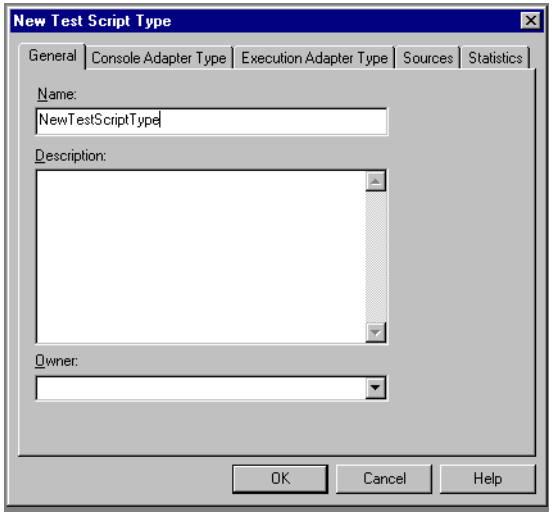
To add a new test script type for Perl test scripts:

1 Create (or designate) a folder for Perl test scripts — for example, C:\testscripts\perl. The folder can be on a local or a network location.

2 From TestManager, click **Tools > Manage > Test Script Types**. The Manage Test Script Types dialog box appears.

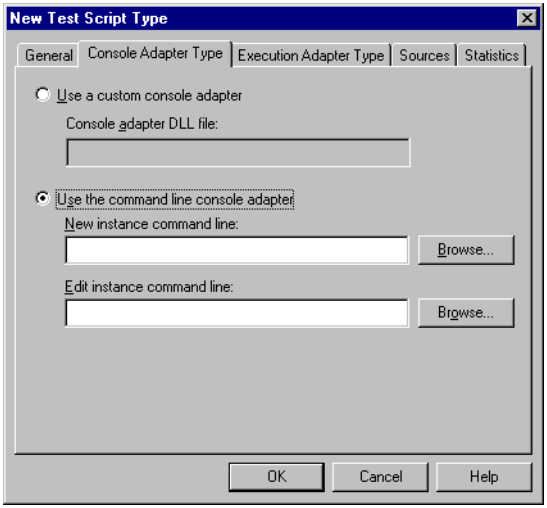


3 Click the **New** button. The New Test Script Type dialog box appears with the **General** tab selected.



In the **Name** box, type the name of the new test script type — for example, `Perl Script`. Optionally, type a description and select an owner. Only the owner can edit or delete this script type.

4 Click the **Console Adapter Type** tab. The dialog box changes as shown below.



Click **Use the command line console adapter** and fill in the boxes as follows:

- In the **New instance command line** box, type the command to execute in order to create a new test script — the name of your favorite editor. For example:

  ```
  notepad
  ```

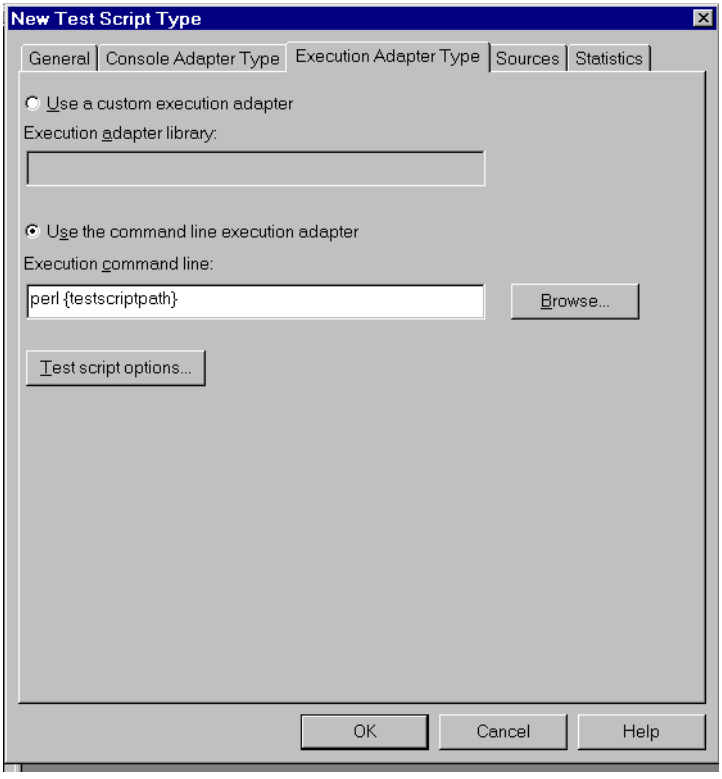
- In the **Edit instance command line** box, type the command to start in order to view or edit existing scripts of this type. For example:

  ```
  notepad {testscriptpath}
  ```

  Type `{testscriptpath}` exactly as shown.

The program you enter (in this case `notepad`) must be in your path.

5 Click the **Execution Adapter Type** tab. The dialog box changes as shown below.

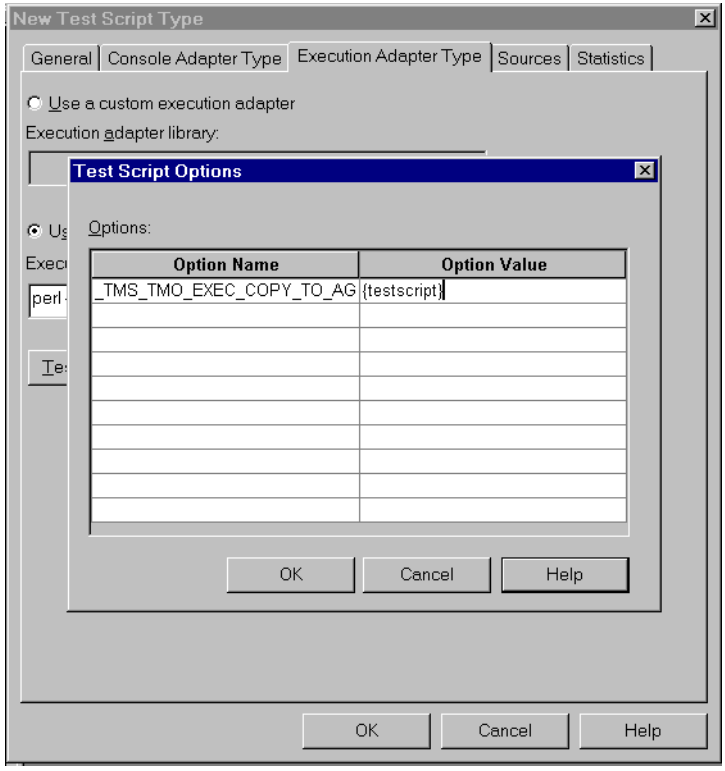


Click **Use the command line execution adapter**. In the **Execution command line** box, type the execution command line for a new script instance. In this example, type the following exactly as shown:

```
perl {testscriptpath}
```

The program (`perl`) must be in your path. (A copy that is released with TestManager is located in the Rational Test folder, which will be in your path by default.)

6 Click **Test Script Options**. The Test Script Options dialog box opens as shown below.

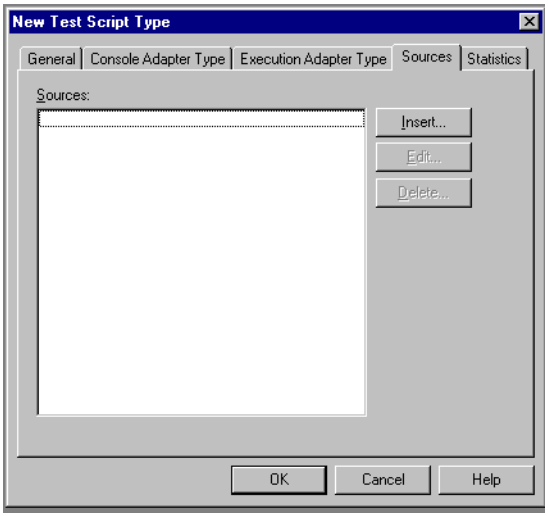


In the **Options** area, type the following Option Name and Option Value pair:

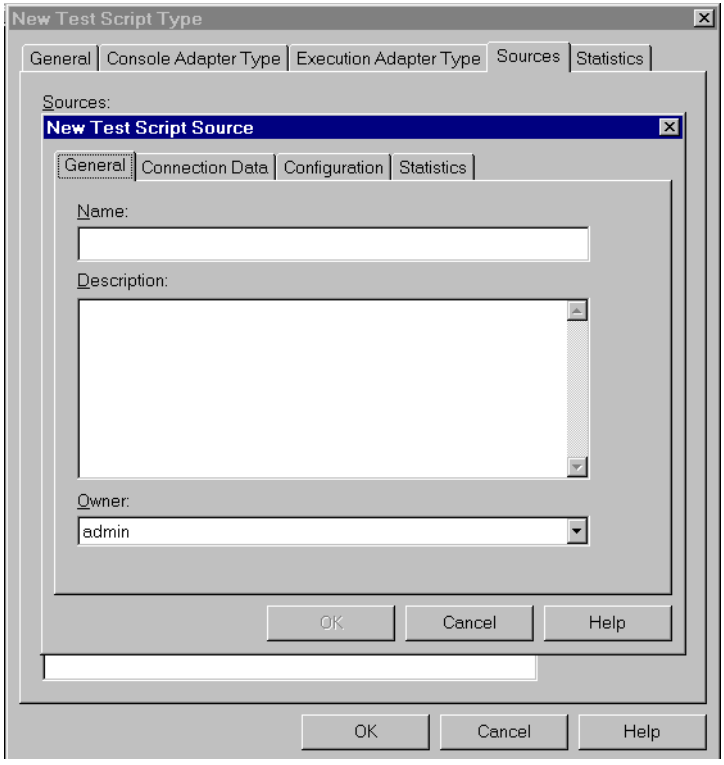
Option Name: `_TMS_TSO_EXEC_COPY_TO_AGENT_FILELIST`
Option Value: `{testscript}`

Click **OK**.

7 Click the **Sources** tab. The dialog box changes as shown below.



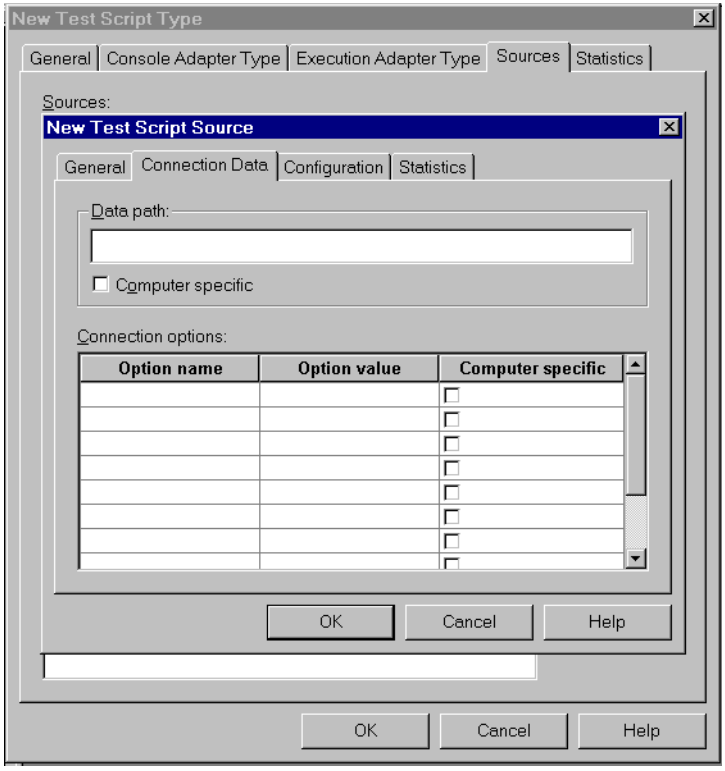
8 Click **Insert**. A popup appears telling you that the test script you are defining must be created before proceeding — answer **Yes**. The dialog box changes as shown below.



In the **Name** box, type a descriptive name for this source. Optionally, type a description and an owner. Only the owner can edit or delete this source.

The **Name** you type here is added to the TestManager **File > New Test Script**, **File > Open Test Script**, and **File > Run Test Script** lists. Select this name to create a new Perl script or edit, view, or run an existing Perl script.

9 Click the **Connection Data** tab. The dialog box changes as shown below.



In the **Data path** box, type the directory name (corresponding to **Name**) that you designated in step 1. This is where source files for test scripts of this type are located.

If the data path might vary from one local computer to another, click **Computer specific**. In this case, the TestManager user is prompted for the actual path of a script at the time of selection.

The **Connection options** box allows you to specify platform-specific execution options for the script type's executable file (in this case, for `perl`). No connection options are needed for this example. Click **OK** and close the dialog box to conclude the procedure.

Setting Up TestManager to Run UNIX Shell Scripts

From TestManager, you can run test scripts (which may contain **tsscmd** statements) written in a UNIX shell language (**sh**, **csh**, **bash**, etc.). The TestManager setup procedure is identical to the Perl setup example described above except for step 5. In the **Execution command line** box, type this execution command line for Bourne shell scripts:

```
tr -d '\r' < {testscriptpath} > | /bin/sh
```

If you create or edit a **sh** script from Windows, it will have a carriage return character at the end of every line. The piped translate (**tr**) command removes this character (\r) if present, so that the script will execute on UNIX. If this character is not present this command has no affect.

With the appropriate software installed, it's possible to run a UNIX shell script on a Windows computer. If you're doing this, omit the translate command shown above and enter instead the Windows executable that interprets **sh** scripts.

If the UNIX shell test scripts must run only on UNIX agents, make sure that the run properties of the User Group that executes the scripts specify UNIX computers. Alternatively, you can create a test configuration to prevent TestManager from running the test scripts on Windows machines: see **configurations:about** in the online Help index.

The designated source directory where shell test scripts are stored (step 9) can be a local Windows folder or an NFS mount of a UNIX filesystem. If the location is a local folder, transfer to this folder the shell scripts you want to run from TestManager.

Using Test Script Options

The **tsscmd** execution adapter uses a number of test script options. These options have initial values, which TestManager users can view or change as explained below. Also, a test script (or other application) can get the current value of an option using `GetScriptOption`.

A TestManager user can also define and set execution options. For example, a user can:

- Define a new test script option named `repeat_count` and assign it the value 3.
- Query the option name from a test script with `GetScriptOption`, and branch based on the returned value of the option.

Test script options can be set at these levels of generality, where 1 is the highest:

1. Test Script Type
2. Test Script Source
3. Test Script
4. Test Script in a suite
5. Test Case implemented by a test script

The Test Script Execution Engine implements identically-named options hierarchically, with lower level settings overriding higher level settings. Thus, if you set the option named `repeat_count` to a different value at each of the levels listed above, the lower level settings override the higher level settings:

- 4 or 5 (mutually exclusive) override 3, 2, and 1
- 3 overrides 2 and 1
- 2 overrides 1

Conversely, if you set an option only at the Test Script Type level, that setting will apply globally for this type of test script. Thus, if you set the option named `repeat_count` to the value 3 and the Test Script Type level, the option will have this value in all instances.

Editing Test Script Options for a Test Script Type

To edit a test script option from Test Manager at the Test Script Type level:

1. From TestManager, click **Tools > Manage > Test Script Type**. The Manage Test Script Type dialog box opens.
2. From the list of existing test script types, click the type whose options you want to edit.
3. Click **Edit**. The Test Script Properties dialog box opens.
4. Click the **Execution Adapter Type** tab.
5. Click **Test Script Options ...** . The Test Script Options dialog box opens. Set or change the desired option values as explained in “Setting or Viewing Option Values” on page 11.

Editing Test Script Options for a Test Script Source

A test script source is a location where designated test scripts are stored. To edit a test script option at the Test Script Source level:

1. Perform steps 1-3 as described above in "Editing Test Script Options for a Test Script Type".
2. From the Test Script Properties dialog, click the **Sources** tab.
3. From the list of sources, click the appropriate source location.
4. Click **View**.
5. Click the **Configuration** tab.
6. Click **Test Script Options ...** . The Test Script Options dialog box opens. Set or change the desired option values as explained in "Setting or Viewing Option Values" on page 11.

Editing Test Script Options for a Test Script

To edit a test script option at the Test Script Asset level:

1. From TestManager, click **View > Test Scripts**. The Test Scripts dialog box opens.
2. Do one of the following:
   a. Right-click a test script type folder and then click **Test Script Options ...**.
   b. Open a test script type folder, browse to a test script, right-click and then click **Test Script Options ...**.

   The Test Script Options dialog box opens. Set or change the desired option values as explained in "Setting or Viewing Option Values" on page 11.

Editing Test Script Options for an Instance in a Suite

To edit a test script option for a script instance in a suite:

1. In TestManager, click **File > Open Suite ...**. The Open Suite dialog box opens.
2. From the list of existing suites, click the suite to open.
3. Click **OK**. The Suite dialog box opens.
4. Open the appropriate user group and browse to a test script.
5. Right-click the test script and then click **Run Properties**. The Run Properties of Test Script dialog box opens.
6. Click **Test Script Options ...** . The Test Script Options dialog box opens. Set or change the desired option values as explained in "Setting or Viewing Option Values" on page 11.

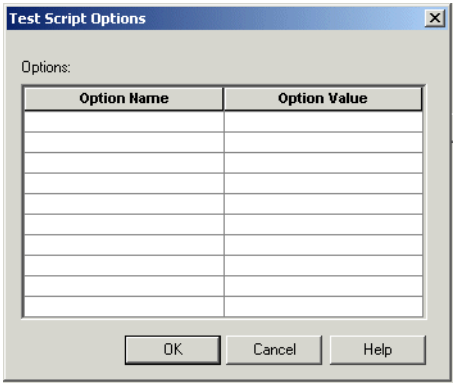
Editing Test Script Options for a Test Case Instance

To edit a test script option for a test case instance:

1. In TestManager, in the Planning view, expand the **Test Plans** folder.
2. Right-click the test plan containing the test case and then click **Open**. The Test Plan dialog box opens.
3. Expand the test cases folder containing the test case you want to edit.
4. Right-click the test case and then click **Properties ...**. The Test Case Properties dialog box opens.
5. Click the **Implementation** tab.
6. Click **Test Script Options ...** . The Test Script Options dialog box opens. Set or change the desired option values as explained in "Setting or Viewing Option Values" on page 11.

Setting or Viewing Option Values

You edit test script options from the Test Script Options dialog box. To open this dialog box, in TestManager, click **View > Test Scripts**. Right-click a test script type, test script source, or test script, and then click **Test Script Options**.



To set a new option value, type its name in the **Option Name** column and its value in the corresponding **Option Value** column, and click **OK**. To change an existing option setting, click the **Option Value** column of the appropriate row, type its value, and click **OK**.

tsscmd Format

tsscmd statements have one of the following two basic formats:

```
tsscmd command options arguments
value = 'tsscmd command options arguments'
```

where:

- *command* is a keyword indicating the Test Script Service you are requesting.
- *options* indicates zero or more options supported by *command*. Option names are preceded by a "-" (hyphen) and might be followed by arguments. If present, options must precede *arguments*.
- *arguments* indicates zero or more values that might be required by *command*. If present, arguments are positional (must be specified in order) and must follow any *options*. Argument strings that contain spaces (or any characters with special meaning to the scripting language, such as ".") must be quoted.

In the second format, *value* is a variable defined in whatever scripting language you are using: the **tsscmd** expression will return a value to this variable, which can then be used in the test script in whatever manner the scripting language allows.

Note that `''` indicate delimiters. Some delimiter is required, but a different delimiter might be used, or required, with different scripting languages. For example, in Perl, here are the correct command formats:

```
'tsscmd command options arguments';
$value = 'tsscmd command options arguments';
```

With the first format (no value returned), you can use the Perl `system` function.

Both *command* and *options* are case-insensitive and can be abbreviated by the shortest unique string. Thus, two statement options named **-access** and **-ascend** can be specified as **-ACCESS**, **-ASCEND**, **-ac**, and **-as**. Similarly, the command **DatapoolOpen** can be entered as **datapoolopen**, **DATAPOOLOPEN**, **datapool0**, **DATAPOOLo**, and so on.

Sample Command Line Test Script

The following example illustrates how to use **tsscmd** statements inside a Perl script. The example opens a datapool and displays some of its attributes. If the datapool fails to open, the script calls `ErrorDetail` for information. If you create a datapool (click **Tools > Manage > Datapools**) matching the name entered in the script's first line, you can run the script from TestManager (click **File > Run Test Script > Command Line**). Or you can

add it to a suite containing test scripts of other types and run the suite. If you follow the procedure explained in *Setting Up TestManager for tsscmd* on page 1, you can write, edit and view this test script from the TestManager **File** menu, as well as run it.

```
$dpid= `tsscmd datapoolopen -access private contacts`;
chomp ($dpid);
$? = $? >> 8;
if ($? == 0) { # datapool is open
    print "Datapool opened: here are some of its attributes\n";
    print "Datapool ID for this run is $dpid\n";
    $ncol= `tsscmd datapoolcolumncount $dpid`;
    print "datapool has $ncol columns\n";
    for ($i=1; $i le $ncol; $++i) {
      $cname= `tsscmd datapoolcolumnname $dpid $i`;
      print "Column $i is named $cname\n";
    }
    $nrows = `tsscmd datapoolrowcount $dpid`;
    print "datapool has $nrows rows\n";
    `tsscmd datapoolclose $dpid`;
  }
else{         # datapool open failed
    print "datapool failed to open with status code $?\n";
    print `tsscmd errordetail`;
}
```

Editing and Storing Test Scripts

To open a test script in TestManager, click **File > Open Test Script**. TestManager opens the test script using the editor you specified when you added the test script type (step 4 in “Setting Up TestManager for tsscmd” on page 1). Test scripts are stored in the folder you indicated when you added the test script type (step 9 in “Setting Up TestManager for tsscmd” on page 1).

To create a test script, click **File > New Test Script**. then select the appropriate type. TestManager starts an editing session with the editor you specified when added the test script type (step 4 in “Setting Up TestManager for tsscmd” on page 1).

When you’ve written your new script, be sure to save it in the folder you specified when you added the test script type (step 9 in “Setting Up TestManager for tsscmd” on page 1).

Running Test Scripts

You can run Command Line test scripts containing **tsscmd** statements either from within the TestManager GUI, or from a command line via the **rttsee** command. You cannot run a Command Line test script containing **tsscmd** statements directly from the command line (by typing the test script's name.)

Running a Test Script from TestManager

This is the usual way to run test scripts containing **tsscmd** statements. You can:

- Run a single test script by itself. If you have added one or more test script types that use the Command Line execution adapter as we recommended (see "Setting Up TestManager for tsscmd" on page 1), you do this by selecting **File > Run Test Script >** *type*, where *type* is the name you chose for the type (such as perl). Then you select the test script you want to run from a list. Alternatively (this is the only choice if you have not added a new test script type), you can select **File > Run Test Script > Command Line** and either type the path name of the script to run or use the browse button.
- Run a test script from within a test case (**File > Run Test Case**).
- Add the test script to a TestManager suite and run the suite (**File > Run Suite**). A suite can include different types of test scripts — for example, you can add Command Line test scripts containing **tsscmd** statements to a suite that also contains Java, Visual Basic, GUI, VU, or custom test script types. For information about adding scripts to a TestManager suite, see the *Using Rational TestManager* manual.

Running a Test Script with rttsee

The **rttsee** program allows you to run a test script through its TSEA from the command line rather than from TestManager. For example, if you add a test script named datapoolTest following the instructions in *Sample Command Line Test Script* on page 12, you can run the script from a Windows command window as explained below.

1. Start a TSS server at a listening port (any port above 1024 will do). For example:

   ```
   rttsee -k -P 3298
   ```

2. Set environment variable `RTTSS_HOST` to localhost and `RTTSS_PORT` to the port number you used in step 1. (On Windows systems, use the System Properties dialog.)

3. Issue the run command. For example:

```
rttsee -e rttseacmd datapoolTest
```

The **rttsee** interface is useful for debugging, and for running test scripts on non-Windows platforms (for example, testing a UNIX Bourne shell script containing **tsscmd** statements). However, scripts that are run via this interface do not have access to the TestManager monitoring and reporting functions, so normally you use **rttsee** only for debugging or during development.

Test scripts are stored in a folder you specified when you added the Command Line test script type: see step 7 in section *Setting Up TestManager for tsscmd* on page 1. TestManager cannot execute test scripts that are stored in an unregistered location.

The syntax of rttsee is:

```
rttsee [option [arg]]
```

The full options are described in the following table.

Option	Description
`-d` *dir*	Specifies the directory for result files — u-file (log), o-file, e-file. The default is the current directory.
`-e` *tsea*[*:type*] *script*[*:type*]	Specifies the TSEA to start and the test script to run. If *tsea* handles test scripts of more than on type, *:type* indicates the type of *script*. The *:type* may be specified with either or both the TSEA or script, but it must match if specified with both.
`-G` [I | i T | t]	Controls random number generation. Enter one choice (`I` or `i`, `T` or `t`) from either or both pairs: ▪ `I` Generate unique seeds for each virtual tester, using either the predefined seed or one specified with `-S` (default). ▪ `i` Use the same seed for all virtual testers, either the predefined seed or one specified with `-S`. ▪ `t` Seed the generator once for all tasks at the beginning, using either the predefined seed or one specified with `-S` (default). ▪ `T` Reseed the generator at the beginning of each task.
`-k`	Keep-alive. Use with `-P` to start a TSS server that keeps running after all test scripts have completed execution.
`-P` *portnumber*	Specifies the listening port for a TSS server that remains alive until explicitely stopped.
`-r`	Redirects stdio to the o-file and e-file (in the directory specified by `-d`).
`-S` *seed*	Specifies an alternative seed value for the predefined seed. Must be a positive integer except in conjunction with `-G i`.
`-u` *uid*	Specifies the ID of a virtual tester.
`-V`	Displays the `rttsee` version.

tsscmd Output

tsscmd statements can deposit information in any of these locations:

- Test log
- Error and output files
- TestManager shared memory

The following sections describe these locations.

Test Log

The test log (or *log*) is where TestManager lists the test cases that have been run and their pass/fail results. TestManager uses the information in the log to generate reports.

You can also write pass/fail results to the log and log messages and errors, using the following comands:

- *LogEvent* on page 33
- *LogMessage* on page 34
- *LogTestCaseResult* on page 36
- *CommandEnd* on page 38
- *CommandStart* on page 39
- *LogCommand* on page 87

For test scripts executed from within TestManager, use the TestManager **ViewLog** button to view the log of test scripts. For test scripts executed outside the TestManager UI (with **rttsee**), the log file is in the current working directory by default but can be redirected by the **-d** and **-r** option switches.

Error File and Output File

As a development and debugging aid, you can write information to an output and an error file using the `Print` and `ePrint` commands, respectively.

For test scripts executed from within TestManager, use the TestManager **perfdata** button to view output and error logs. For test scripts executed outside the TestManager UI (with **rttsee**), the output and error files are in the current working directory by default but can be redirected by the **-d** and **-r** option switches.

TestManager Shared Memory

Shared memory is used to provide data for the TestManager runtime console, and to pass information among test scripts during playback.

To write data to shared memory, use the methods described in the following sections:

- *Monitor Commands* on page 68. These commands provide TestManager with data needed for monitoring operations.
- *Synchronization Commands* on page 76. These commands allow concurrently running scripts to share data.

Error Handling

If an error occurs in a script, the script stops running and (ususally) TestManager generates an error file. However, for command line test scripts (including those containing **tsscmd** statements), TestManager does not log a Fail result for scripts that fail. Your script is responsible for error checking and handling.

All **tsscmd** statements return numeric status codes, which are documented with each statement. In addition, many return values as well. For example, when successful `SharedVarWait` returns:

- The value of the specified shared variable before the adjustment is performed.
- A status code of 0 or 1 indicating whether or not the value of the shared variable reached a specified range within a specified timeout interval

On failure, `SharedVarWait` returns one of three integers (4, 5, 8) indicating the cause of the failure. The following fragment indicates how you could check for status return codes and obtain additional information about a failure in Perl.

```
$before = ‘tsscmd SharedVarWait -t 60000 svFoo 10 20’;
$? = $? >> 8;
if ($? == 0) {
   ‘tsscmd LogMessage timeout expired, value was $before‘;
}
elsif ($? == 1) {
   ‘tsscmd LogMessage condition was met before timeout expired‘;
}
else {
   ‘tsscmd LogMessage unexpected exit status $?‘;
   $detail = ‘tsscmd ErrorDetail’;
   chomp ($detail);
   ‘tsscmd LogMessage $detail‘;
}
```

Limitation

Test scripts which have more than one virtual tester, and which use datapools, synchronization points, or shared variables, will not run on agents. The scripts will run on the local (TestManager) host.

A workaround to this limitation exists: run, in the same test suite, a VU script that declares the same datapools, synchronization points, and shared variables.

Test Script Services Reference 2

About Test Script Services

This chapter describes the Rational Test Script Services (TSS). It explains the **tsscmd** commands you use to give test scripts access to services such as datapools, measurement, virtual tester synchronization, and monitoring.The commands are divided into the following functional categories.

Category	Description
Datapool	Provide variable data to test scripts during playback.
Logging	Log messages for reporting and analysis.
Measurement	Manage timers and test variables.
Utility	Perform common test script functions.
Monitor	Monitor test script playback progress.
Synchronization	Synchronize virtual testers in multicomputer runtime environments.
Session	Manage the test suite runtime environment.
Advanced	Perform advanced logging and measurement functions.

Datapool Commands

During testing, it is often necessary to supply an application with a range of test data. Thus, in the functional test of a data entry component, you may want to try out the valid range of data, and also to test how the application responds to invalid data. Similarly, in a performance test of the same component, you may want to test storage and retrieval components in different combinations and under varying load conditions.

A *datapool* is a source of data stored in a Rational project that a test script can draw upon during playback, for the purpose of varying the test data. You create datapools from TestManager, by clicking **Tools > Manage > Datapools**. For more information, see the datapool chapter in the *Rational TestManager User's Guide*. Optionally, you can import manually created datapool information stored in flat ASCII Comma Separated Values (CSV) files, where a row is a newline-terminated line and columns are fields in the line separated by commas (or some other field-delimiting character).

Summary

Use the datapool commands listed in the following table to access and manipulate datapools within your scripts.

Command	Description
`DatapoolClose`	Closes a datapool.
`DatapoolColumnCount`	Returns the number of columns in a datapool.
`DatapoolColumnName`	Returns the name of the specified datapool column.
`DatapoolFetch`	Moves the datapool cursor to the next row.
`DatapoolOpen`	Opens the named datapool and sets the row access order.
`DatapoolRewind`	Resets the datapool cursor to the beginning of the datapool access order.
`DatapoolRowCount`	Returns the number of rows in a datapool.
`DatapoolSearch`	Searches a datapool for the named column with a specified value.
`DatapoolSeek`	Moves the datapool cursor forward.
`DatapoolValue`	Retrieves the value of the specified datapool column.

DatapoolClose

Closes a datapool.

Syntax

```
tsscmd DatapoolClose dpid
```

Element	Description
dpid	The ID of the datapool to close. Returned by `DatapoolOpen`.

Return Value

This command exits with one of the following results:

- 0 –Success.
- 4 – Server connection failure.
- 5 – The datapool identifier is invalid.

Example

This example opens the datapool `custdata` with default row access and closes it.

```
dpid = 'tsscmd DatapoolOpen custdata'
tsscmd DatapoolClose dpid
```

See Also

`DatapoolOpen`

DatapoolColumnCount

Returns the number of columns in a datapool.

Syntax

```
columns = 'tsscmd DatapoolColumnCount dpid'
```

Element	Description
dpid	The ID of the datapool. Returned by `DatapoolOpen`.

Return Value

On success, this command returns the number of columns in the specified datapool. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 5 – The datapool identifier is invalid.
- 8 – Pending abort resulting from a user request to stop a suite run.

Example

This example opens the datapool `custdata` and gets the number of columns.

```
dpid = 'tsscmd DatapoolOpen custdata'
columns = 'tsscmd DatapoolColumnCount dpid'
```

DatapoolColumnName

Gets the name of the specified datapool column.

Syntax

```
columnName = 'tsscmd DatapoolColumnName dpid columnNumber'
```

Element	Description
dpid	The ID of the datapool. Returned by `DatapoolOpen`.
columnNumber	A positive number indicating the number of the column whose name you want to retrieve. The first column is number 1.

Return Value

On success, this command returns the name of the specified datapool column. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 5 – The datapool identifier or column number is invalid.
- 8 – Pending abort resulting from a user request to stop a suite run.

Example

This example opens a three-column datapool and gets the name of the third column.

```
dpid = 'tsscmd DatapoolOpen custdata'
tsscmd DatapoolFetch dpid
colName = 'tsscmd DatapoolColumnName dpid 3'
```

DatapoolFetch

Moves the datapool cursor to the next row.

Syntax

```
tsscmd DatapoolFetch dpid
```

Element	Description
dpid	The ID of the datapool. Returned by DatapoolOpen.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 3 – The end of the datapool was reached.
- 4 – Server connection failure.
- 5 – The datapool identifier is invalid.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

This call positions the datapool cursor on the next row and loads the row into memory. To access a column of data in the row, call DatapoolValue.

The "next row" is determined by the *assessFlags* passed with the open call. The default is the next row in sequence. See DatapoolOpen.

After a datapool is opened, a DatapoolFetch is required before the initial row can be accessed.

An end-of-file condition results if a script fetches past the end of the datapool, which can occur only if access flag `NOWRAP` was set on the open call. If the end-of-file condition occurs, the next call to `DatapoolValue` results in a runtime error.

Example

This example opens datapool `custdata` with default (sequential) access and positions the cursor to the first row.

```
dpid = 'tsscmd DatapoolOpen custdata'
tsscmd DatapoolFetch dpid
```

See Also

`DatapoolOpen`, `DatapoolSeek`, `DatapoolValue`

DatapoolOpen

Opens the named datapool and sets the row access order.

Syntax

```
dpid ='tsscmd DatapoolOpen [-access accessFlags] name
   [colname=value...]'
```

Element	Description
name	The name of the datapool to open. If *accessFlags* includes `NO_OPEN`, no CSV datapool is opened; instead, *name* refers to the specified name/value pairs specifying a one-row table. Otherwise, the CSV file *name* in the Rational project is opened.

Element	Description
accessFlags	Optional flags indicating how the datapool is accessed when a script is played back. Specify at most one value from each of the following categories: **1** Specify the sequence in which datapool rows are accessed: SEQUENTIAL – Physical order (default) RANDOM – Any order, including multiple access or no access SHUFFLE – Access order is shuffled after each access **2** Specify what happens after the last datapool row is accessed: NOWRAP – End access to the datapool (default) WRAP – Go back to the beginning **3** Specify whether the datapool cursor is shared by all virtual testers or is unique to each: PRIVATE – Virtual testers each work from their own sequential, random, or shuffle access order (default) SHARED – All virtual testers work from the same access order **4** PERSIST specifies that the datapool cursor is persistent across multiple script runs. For example, with a persistent cursor, if the row number after a suite run is 100, the first row accessed in a subsequent run is numbered 101. Cannot be used with PRIVATE. Ignored if used with RANDOM. **5** REWIND specifies that the datapool should be rewound when opened. Ignored unless used with PRIVATE. **6** NO_OPEN specifies that, instead of a CSV file, the opened datapool consists only of specified column/value pairs.
colname=value ...	Optionally, a list of one or more column/value pairs, where *colname* is the column name and *value* is the override value to be returned by DatapoolValue for that column name.

Return Value

On success, this command returns a positive integer indicating the ID of the opened datapool. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 5 – The *accessFlags* argument is or results in an invalid combination.
- 7 – No datapool of the given *name* was found.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

If the *accessFlags* argument is specified as 0 or omitted, the rows are accessed in the default order: sequentially, with no wrapping, and with a private cursor. If multiple *accessFlags* are specified, they must be valid combinations as explained in the syntax table.

If you close and then reopen a private-access datapool with the same *accessFlags* and in the same or a subsequent script, access to the datapool is resumed as if it had never been closed.

If multiple virtual testers access the same datapool in a suite, the datapool cursor is managed as follows:

- The first open that uses the `SHARED` option initializes the cursor. In the same suite run (and, with the `PERSIST` flag, in subsequent suite runs), virtual testers that subsequently use the same datapool opened with `SHARED` share the initialized cursor.
- The first open that uses the `PRIVATE` option initializes the private cursor for a virtual tester. In the same suite run, a subsequent open that uses `PRIVATE` sets the cursor to the last row accessed by that virtual tester.

Example

This example opens the datapool named `custdata`, with a modified row access.

```
dpid = 'tsscmd DatapoolOpen -a SHUFFLE -a PERSIST custdata'
```

See Also

`DatapoolClose`

DatapoolRewind

Resets the datapool cursor to the beginning of the datapool access order.

Syntax

```
tsscmd DatapoolRewind dpid
```

Element	Description
dpid	The ID of the datapool. Returned by `DatapoolOpen`.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 5 – The datapool identifier is invalid.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

The datapool is rewound as follows:

- For datapools opened `SEQUENTIAL`, `DatapoolRewind` resets the cursor to the first record in the datapool file.
- For datapools opened `RANDOM` or `SHUFFLE`, `DatapoolRewind` restarts the random number sequence.
- For datapools opened `SHARED`, `DatapoolRewind` has no effect.

At the start of a suite, datapool cursors always point to the first row.

If you rewind the datapool during a suite run, previously accessed rows are fetched again.

Example

This example opens the datapool `custdata` with default (sequential) access, moves the access to the second row, and then resets access to the first row.

```
dpid = 'tsscmd DatapoolOpen custdata'
tsscmd DatapoolSeek dpid 2
tsscmd DatapoolRewind dpid
```

DatapoolRowCount

Returns the number of rows in a datapool.

Syntax

```
rows = 'tsscmd DatapoolRowCount dpid'
```

Element	Description
dpid	The ID of the datapool. Returned by `DatapoolOpen`.

Return Value

On success, this command returns the number of rows in the specified datapool. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 5 – The datapool identifier is invalid.
- 8 – Pending abort resulting from a user request to stop a suite run.

Example

This example opens the datapool `custdata` and gets the number of rows in the datapool.

```
dpid = 'tsscmd DatapoolOpen custdata'
rows = 'tsscmd DatapoolRowCount dpid'
```

DatapoolSearch

Searches a datapool for a named column with a specified value.

Syntax

```
tsscmd DatapoolSearch dpid column=value [...]
```

Element	Description
dpid	The ID of the datapool. Returned by `DatapoolOpen`.
column=value	One or more column/value pairs to be searched for.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 3 – The end of the datapool was reached.
- 4 – Server connection failure.
- 5 – The datapool identifier is invalid.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

When a row is found containing the specified values, the cursor is set to that row.

Example

This example searches the datapool `custdata` for a row containing the column named `Last` with the value `Doe`:

```
dpid = 'tsscmd DatapoolOpen custdata'
rowNumber='tsscmd DatapoolSearch dpid Last=Doe'
```

DatapoolSeek

Moves the datapool cursor forward.

Syntax

```
tsscmd DatapoolSeek dpid count
```

Element	Description
dpid	The ID of the datapool. Returned by DatapoolOpen.
count	A positive number indicating the number of rows to move forward in the datapool.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 3 – The end of the datapool was reached.
- 4 – Server connection failure.
- 5 – The datapool identifier is invalid.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

This call moves the datapool cursor forward *count* rows and loads that row into memory. To access a column of data in the row, call DatapoolValue.

The meaning of "forward" depends on the *accessFlags* passed with the open call; see DatapoolOpen. This call is functionally equivalent to calling DatapoolFetch *count* times.

An end-of-file error results if cursor wrapping is disabled (by access flag NOWRAP) and *count* moves the access row beyond the last row. If DatapoolValue is then called, a runtime error occurs.

Example

This example opens the datapool `custdata` with the default (sequential) access and moves the cursor forward two rows.

```
dpid = 'tsscmd DatapoolOpen custdata'
tsscmd DatapoolSeek dpid 2
```

See Also

`DatapoolFetch`, `DatapoolOpen`, `DatapoolValue`

DatapoolValue

Retrieves the value of the specified datapool column in the current row.

Syntax

`value` = 'tsscmd **`DatapoolValue`** *`dpid columnName`*'

Element	Description
`dpid`	The ID of the datapool. Returned by `DatapoolOpen`.
`columnName`	The name of the column whose value you want to retrieve.

Return Value

On success, this command returns the value of the specified datapool column in the current row. The command exits with one of the following results:

- 0 – Success.
- 3 – The end of the datapool was reached.
- 4 – Server connection failure.
- 5 – The specified *`columnName`* is not a valid column in the datapool.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

This call gets the value of the specified datapool column from the current datapool row, which has been loaded into memory either by `DatapoolFetch` or `DatapoolSeek`.

By default, the returned value is a column from a CSV datapool file located in a Rational datastore. If the datapool open call included the `NO_OPEN` access flag, the returned value comes from an override list provided with the open call.

This method	Generates
`booleanValue()`	The `boolean` representation of the datapool value.
`byteValue()`	The byte representation of the datapool value.
`charValue()`	The character representation of the datapool value.
`doubleValue()`	The `double` representation of the datapool value.
`floatValue()`	The `float` representation of the datapool value.
`getBigDecimal()`	The `BigDecimal` representation of the datapool value.
`intValue()`	The `int` representation of the datapool value.
`longValue()`	The `long` representation of the datapool value.
`shortValue()`	The `short` representation of the datapool value.
`toString()`	The `String` representation of the datapool value.

Example

This example retrieves the value of the column named `Middle` in the first row of the datapool `custdata`.

```
dpid = 'tsscmd DatapoolOpen custdata'
tsscmd DatapoolFetch dpid
colVal = 'tsscmd DatapoolValue dpid Middle'
```

See Also

`DatapoolFetch`, `DatapoolOpen`, `DatapoolSeek`

Logging Commands

Use the logging commands to build the log that TestManager uses for analysis and reporting. You can log events, messages, or test case results.

A logged event is the record of something that happened. Use the environment variable `LogEvent_control` (page 92) to control whether or not an event is logged.

An event that gets logged may have associated data (either returned by the server or supplied with the statement). Use the environment variable `LogData_control` (page 92) to control whether or not any data associated with an event is logged.

Summary

Use the commands listed in the following table to write to the TestManager log.

Command	Description
`LogEvent`	Logs an event.
`LogMessage`	Logs a message event.
`LogTestCaseResult`	Logs a test case event.

LogEvent

Logs an event.

Syntax

```
tsscmd LogEvent [-result result] [-desc description] eventType
   [property=value ...]
```

Element	Description
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ `NONE` (default: no notification) ▪ `PASS` ▪ `FAIL` ▪ `WARN` ▪ `STOPPED` ▪ `INFO` ▪ `COMPLETED` ▪ `UNEVALUATED`
description	Contains the string to be put in the entry's failure description field.
eventType	Contains the description to be displayed in the log for this event.

Element	Description
property=value	Specifies one or more property-value pairs.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 5 – An unknown *result* was specified.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

The event and any data associated with it are logged only if the specified `result` preference matches associated settings in the `LogData_control` (page 92) or `LogEvent_control` (page 92) environment variables. Alternatively, the logging preference can be set with the `Log_level` (page 93) and `Record_level` (page 94) environment variables. The STOPPED, COMPLETED, and UNEVALUATED preferences are intended for internal use.

Example

This example logs the beginning of an event of type `Login Dialog`.

```
tsscmd LogEvent -d "Login script failed" "Login Dialog"
ScriptName=Login LineNumber=1
```

LogMessage

Logs a message.

Syntax

```
tsscmd LogMessage [-result result] [-desc description] message
```

Element	Description
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ NONE (default: no notification) ▪ PASS ▪ FAIL ▪ WARN ▪ STOPPED ▪ INFO ▪ COMPLETED ▪ UNEVALUATED
description	Specifies the string to be put in the entry's failure description field.
message	Specifies the string to log.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

An event and any data associated with it are logged only if the specified *result* preference matches associated settings in the LogData_control (page 92) or LogEvent_control (page 92) environment variables.

Alternatively, the logging preference can be set with the Log_level (page 93) and Record_level (page 94) environment variables. The STOPPED, COMPLETED, and UNEVALUATED preferences are intended for internal use.

Example

This example logs the following message: --Beginning of timed block T1--.

```
tsscmd LogMessage "--Beginning of timed block T1--"
```

LogTestCaseResult

Logs a test case result.

Syntax

```
tsscmd LogTestCaseResult [-result result] [-desc description]
   testcase [property=value ...]
```

Element	Description
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ NONE (default: no notification) ▪ PASS ▪ FAIL ▪ WARN ▪ STOPPED ▪ INFO ▪ COMPLETED ▪ UNEVALUATED
description	Contains the string to be displayed in the event of a log failure.
testcase	Identifies the test case whose result is to be logged.
property=value	Optionally a list of one or more property name/value pairs.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

A test case is a condition, specified in a list of property name/value pairs, that you are interested in. This command searches for the test case and logs the result of the search.

An event and any data associated with it are logged only if the specified *result* preference matches associated settings in the LogData_control (page 92) or LogEvent_control (page 92) environment variables. Alternatively, the logging preference may be set by the Log_level (page 93) and Record_level (page 94) environment variables. The STOPPED, COMPLETED, and UNEVALUATED preferences are intended for internal use.

Example

This example logs the result of a test case named Verify login.

```
tsscmd TestCaseResult "Verify login" Result=OK
```

Measurement Commands

Use the measurement commands to set timers and environment variables to get the value of internal variables. Timers allow you to gauge how much time is required to complete specific activities under varying load conditions. Environment variables allow for the setting and passing of information to virtual testers during script playback. Internal variables store information used by the TestManager to initialize and reset virtual tester parameters during script playback.

Summary

The following table lists the measurement commands.

Command	Description
CommandEnd	Logs an end-command event.
CommandStart	Logs a start-command event.
EnvironmentOp	Sets an environment variable.
GetTime	Gets the elapsed time of a run.
InternalVarGet	Gets the value of an internal variable.
Think	Sets a think-time delay.
TimerStart	Marks the start of a block of actions to be timed.
TimerStop	Marks the end of a block of timed actions.

CommandEnd

Marks the end of a timed command.

Syntax

```
tsscmd CommandEnd [-desc description] [-start starttime] [-end
   endtime] result logdata [property=value ...]
```

Element	Description
description	Contains the string to be displayed in the event of failure.
starttime	An integer indicating a time stamp to override the time stamp set by `CommandStart`. To use the time stamp set by `CommandStart`, omit or specify as `0`.
endtime	An integer indicating a time stamp to override the current time. To use the current time, omit or specify as `0`.
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ `NONE` (default: no notification) ▪ `PASS` ▪ `FAIL` ▪ `WARN` ▪ `STOPPED` ▪ `INFO` ▪ `COMPLETED` ▪ `UNEVALUATED`
logdata	Text to be logged describing the ended command.
property=value	Optionally specify one or more property name/value pairs.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

The command name and label entered with `CommandStart` are logged, and the run state is restored to the value that existed before the `CommandStart`call.

An event and any data associated with it are logged only if the specified *`result`* preference matches associated settings in the `LogData_control` (page 92) or `LogEvent_control` (page 92) environment variables. Alternatively, the logging preference can be set with the `Log_level` (page 93) and `Record_level` (page 94) environment variables. The STOPPED, COMPLETED, and UNEVALUATED preferences are intended for internal use.

Example

This example marks the end of the timed activity specified by the previous `CommandStart` call.

```
tsscmd CommandEnd -d "Command timer failed" PASS "Login command
completed"
```

See Also

`CommandStart, LogCommand`

CommandStart

Starts a timed command.

Syntax

`tsscmd` **`CommandStart`** *`label name state`*

Element	Description
`label`	The name of the timer to be started and logged, or `NULL` for an unlabeled timer.
`name`	The name of the command to time.
`state`	The run state to log with the timed command. See the run state table starting on page 73. You can enter 0 (`MST_UNDEF`) if you're uninterested in the run state.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

A *command* is a user-defined name appearing in the log of a test run. By placing `CommandStart` and `CommandEnd` calls around a block of lines in a script, you can log the time required to complete the actions in the block.

During script playback, TestManager displays progress for different virtual testers. What is displayed for a group of actions associated by `CommandStart` depends on the run state argument. Run states are listed in the run state table starting on page 73.

`CommandStart`increments `cmdcnt`, sets the name, label, and run state for TestManager, and sets the beginning time stamp for the log entry. `CommandEnd` restores the TestManager run state to the run state that was in effect immediately before `CommandStart`.

Example

This example starts timing the period associated with the string `Login`.

```
tsscmd CommandStart -l initTimer Login WAITRESP
```

See Also

`CommandEnd, LogCommand`

EnvironmentOp

Sets a virtual tester environment variable.

Syntax

```
tsscmd EnvironmentOp envVar envOp [envVal]
```

Element	Description
envVar	The environment variable to operate on. See "Arguments of EnvironmentOp" on page 91 for a list and description of environment variable constants.
envOp	The operation to perform. See "Arguments of EnvironmentOp" on page 91 for a list and description of the operation constants..
envVal	The value operated on as specified by *envOp* to produce the new value for *envVar*.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 5 – The timer label is invalid, or there is no unlabeled timer to stop.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

Environment variables define and control the environment of virtual testers. Using environment variables allows you to test different assumptions or runtime scenarios without re-writing your test scripts. For example, you can use environment variables to specify:

- A virtual tester's average think time, the maximum think time, and how the think time is mathematically distributed around a mean value.
- How long to wait for a response from the server before timing out.
- The level of information that is logged and available to reports.

See "Arguments of EnvironmentOp" on page 91 for a list and description of the values that can be used for argument *envVar*.

Environment control options allow a script to control a virtual tester's environment by operating on the environment variables. Every environment variable has, instead of a single value, a group of values: a default value, a saved value, and a current value.

- **default** – The value of an environment variable before any commands are applied to it. Environment variables are automatically initialized to a default value, and, like persistent variables, retain their values across scripts. The `reset` command resets the default value, as listed in the following table.
- **saved** – The saved value of an environment variable can be used as one way to retain the present value of the environment variable for later use. The `save` and `restore` commands manipulate the saved value.
- **current** – TSS supports a last-in-first-out "value stack" for each environment variable. The current value of an environment variable is simply the top element of that stack. The current value is used by all of the commands. The `push` and `pop` commands manipulate the stack.

See the table on page 97 for the values that can be used for argument *envOp*.

Example

This example gets the current value of `Think_dist`. For a more extensive illustration of environment variable manipulation, see "Example: Manipulating Environment Variables" on page 97.

```
tsscmd environmentOp Think_dist eval $cur_dist
```

GetTime

Gets the elapsed time since the beginning of a suite run.

Syntax

```
time=`tsscmd GetTime`
```

Return Value

On success, this command returns the number of milliseconds elapsed in a suite run. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

For execution within TestManager, this call retrieves the time elapsed since the start time shared by all virtual testers in all test scripts in a suite.

For a test script executed outside TestManager, the time returned is the milliseconds elapsed since the start of the `rttsee` process running the script.

Example

This example stores the elapsed time in *`etime`*.

```
etime = `tsscmd GetTime`
```

InternalVarGet

Gets the value of an internal variable.

Syntax

```
ivVal=`tsscmd InternalVarGet internVar`
```

Element	Description
internVar	The internal variable to operate on. See "Arguments of InternalVarGet" on page 99 for a list and description of the internal variable constants..

Return Value

On success, this command returns the value of the specified internal variable. In addition, it returns one of the following values:

- 0 – Success.
- 4 – Server connection failure.
- 5 – The timer label is invalid, or there is no unlabeled timer to stop.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

Internal variables contain detailed information that is logged during script playback and used for performance analysis reporting. This function allows you to customize logging and reporting detail.

Example

This example stores the current value of the `error` internal variable in `IVVal`.

```
IVVal = `tsscmd InternalVarGet error`
```

Think

Puts a time delay in a script that emulates a pause for thinking.

Syntax

```
tsscmd Think [thinkAverage]
```

Element	Description
thinkAverage	If specified as 0 , the number of milliseconds stored in the `Think_avg` environment variable is used as the basis of the calculation. Otherwise, the calculation is based on the value specified.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

A think-time delay is a pause inserted in a performance test script in order to emulate the behavior of actual application users.

For a description of environment variables, see `EnvironmentOp` on page 40.

Example

This example calculates a pause based on the value stored in the environment variable `Think_avg` and inserts the pause into the script.

```
tsscmd Think
```

See Also

`ThinkTime`

TimerStart

Marks the start of a block of actions to be timed.

Syntax

```
tsscmd TimerStart [-label label] [-time timeStamp]
```

Element	Description
label	The name of the timer to be inserted into the log. If specified as NULL or not specified, an unlabeled timer is created. Only one unlabeled timer is supported at a time.
timeStamp	An integer specifying a time stamp to override the current time. If specified as 0 or not specified, the current time is logged.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

This call associates a starting time stamp with *label* for later reference by TimerStop. The TestManager reporting system uses captured timing information for performance analysis reports.

Starting an unlabeled timer sets a start time for an event that you want to subdivide into timed intervals. See the example for TimerStop. You can get a similar result using named timers, but there will be a slight difference in the timing calculation due to the overhead of starting a timer.

Example

This example times actions designated event1, logging the current time.

```
tsscmd TimerStart -l event1
/* action to be timed */
tsscmd TimerStop -l event1
```

See Also

`TimerStop`

TimerStop

Marks the end of a block of timed actions.

Syntax

`tsscmd` **`TimerStop`** `[-remove] [-time` *`timeStamp`*`] label`

Element	Description
`label`	The name to be logged.
`timeStamp`	An integer indicating the time stamp to log. If not specified or specified as 0, the current time is used.
`-r`	Specify to stop and remove the timer or omit to stop the timer without removing it. A timer that is not removed can be stopped multiple times in order to measure intervals of this timed event.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 5 – The timer label is invalid, or there is no unlabeled timer to stop.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

Normally, this call associates an ending time stamp with a label specified with `TimerStart`. If the specified *`label`* was not set by a previous `TimerStart` but an unlabeled timer exists, this call logs an event using the specified label and the start time specified for the unlabeled timer with `TimerStart`. If -r is not specified, multiple invocations of `TimerStop` are allowed against a single `TimerStart`. This usage (see the example) allows you to subdivide a timed event into separate timed intervals.

Example

This example stops an unlabeled timer without removing it. In the log, `event1` and `event2` will record the time elapsed since the `TimerStart` call.

```
tsscmd TimerStart
/* action to be timed */
tsscmd TimerStop -l event1
/* another action to be timed */
tsscmd TimerStop -l event2
```

See Also

`TimerStart`

Utility Commands

Use the utility commands to perform actions common to many test scripts.

Summary

The following table lists the utility commands.

Command	Description
`ApplicationPid`	Gets the process ID of an application.
`ApplicationStart`	Starts an application.
`ApplicationWait`	Waits for an application to terminate.
`Delay`	Delays the specified number of milliseconds.
`ErrorDetail`	Retrieves error information about a failure.
`GetComputerConfiguration AttributeList`	Gets the list of computer configuration attributes and their values.
`GetComputerConfiguration AttributeValue`	Gets the value of a computer configuration attribute.
`GetPath`	Gets a pathname.
`GetScriptOption`	Gets the value of a script playback option.
`GetTestCaseConfiguration Attribute`	Gets the value of a test case configuration attribute.

Command	Description
`GetTestCaseConfiguration AttributeList`	Gets the list of test case configuration attributes and their values.
`GetTestCaseConfigurationName`	Gets the name of the configuration (if any) associated with the current test case.
`GetTestCaseName`	Gets the name of the test case in use.
`GetTestToolOption`	Gets a test case tool option.
`JavaApplicationStart`	Starts a Java application.
`NegExp`	Gets the next negative exponentially distributed random number with the specified mean.
`Rand`	Gets the next random number.
`SeedRand`	Seeds the random number generator.
`StdErrPrint`	Prints a message to the virtual tester's error file.
`StdOutPrint`	Prints a message to the virtual tester's output file.
`Uniform`	Gets the next uniformly distributed random number in the specified range.
`UniqueString`	Returns a unique text string.

ApplicationPid

Gets the process ID of an application.

Syntax

```
pid = 'tsscmd ApplicationPid appHandle'
```

Element	Description
appHandle	The ID of the application whose PID you want to get. Returned by `ApplicationStart` or `JavaApplicationStart`.

Return Value

On success, this command returns the system process ID of the specified application. It exits with one of the following values :

- 0 – Success.
- 5 – The application handle is invalid.

Comments

This command works for applications started by `ApplicationStart` or `JavaApplicationStart`.

A successful invocation does not imply that the application whose PID is returned is still alive nor guarantee that the application is still running under this PID.

Example

This example returns the PID of application `myApp`.

```
myAppHandle = `tsscmd ApplicationStart myApp`
myAppPID = `tsscmd ApplicationPid myAppHandle`
```

See Also

`ApplicationStart, ApplicationWait,JavaApplicationStart`

ApplicationStart

Starts an application.

Syntax

handle = `tsscmd` **`ApplicationStart`** `[-workdir` *workingDir*`]` *appHandle*`

Element	Description
appHandle	The pathname of the application to be started, which can include options and arguments. The file suffix can be omitted.
workingDir	The directory in which to start the application. The current directory if not specified.

Return Value

On success, this command returns a handle for the started application. It exits with one of the following values:

- 0 – Success.
- 5 – The application handle is invalid.

Comments

Example

This example starts application `myApp`.

```
myAppHandle = 'tsscmd ApplicationStart myApp'
```

See Also

`ApplicationPid, ApplicationWait,JavaApplicationStart`

ApplicationWait

Waits for an application to terminate.

Syntax

`tsscmd` **`ApplicationWait`** `[-timeout` *`msec`*`]` *`app`*

Element	Description
app	The application that you are waiting for. Returned by `ApplicationStart` or `JavaApplicationStart`.
msec	The number of milliseconds to wait for *app* to terminate or 0 to return immediately.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 2 – The application was still running when the time-out expired.
- 4 – Server connection failure.

- 6 – The system returned an error: call `ErrorDetail` for information.
- 7 – The process indicated by *app* was not found. It may have terminated before this call or *app* may be an invalid handle.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

This command works for applications started by `ApplicationStart` or `JavaApplicationStart`.

Example

This example waits 600 milliseconds for application `myApp` to terminate.

```
myAppHandle = 'tsscmd ApplicationStart myApp'
tsscmd ApplicationWait -timeout 600 myAppHandle
```

See Also

`ApplicationPid, ApplicationStart,JavaApplicationStart`

Delay

Delays script execution for the specified number of milliseconds.

Syntax

`tsscmd` **`Delay`** *`msecs`*

Element	Description
`msecs`	The number of milliseconds to delay script execution.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

The delay is scaled as indicated by the contents of the `Delay_dly_scale` environment variable. The accuracy of the time delayed is subject to operating system limitations.

Example

This example delays execution for 10 milliseconds.

```
tsscmd Delay 10
```

ErrorDetail

Retrieves error information about a failure.

Syntax

```
errorText='tsscmd ErrorDetail'
```

Return Value

This command returns 0 if the previous command succeeded. If the previous command failed, ErrorDetail returns one of the error codes listed below and corresponding *errorText*.

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Example

This example opens a datapool and, if there is an error, displays the associated error message text.

```
dpid = 'tsscmd DatapoolOpen custdata'
errorText = 'tsscmd ErrorDetail'
```

GetComputerConfigurationAttributeList

Gets the list of computer configuration attributes and their values.

Syntax

```
config = 'tsscmd GetComputerConfigurationAttributeList
   [-single]'
```

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

You create and maintain computer configuration attributes from TestManager. This command returns the current settings.

The computer configuration attribute list can be obtained in either of two formats:

- Without the `-single` option, two result lines are returned for each row with the configuration name appearing on the first and its value on the second.
- With the `-single` option, all rows are returned on one result line containing all pairs in the form `name=value`.

Example

This example returns the current computer configuration attribute list.

```
config = 'tsscmd GetComputerConfigurationAttributeList'
```

See Also

`GetComputerConfigurationAttributeValue`

GetComputerConfigurationAttributeValue

Gets the value of computer configuration attribute.

Syntax

```
value = 'tsscmd GetComputerConfigurationAttributeValue name'
```

Element	Description
name	The name of the computer configuration attribute whose value is to be returned.

Return Value

On success, this command returns a handle for the started application. It exits with one of the following values.

- 0 – Success.
- 4 – Server connection failure.

Example

This example returns the value of the configuration attribute `Operating System`.

```
OSVal = 'tsscmd GetComputerConfigurationAttributeValue "Operating
System"'
```

See Also

`GetComputerConfigurationAttributeList`

GetPath

Gets the root path of a test asset.

Syntax

```
value = 'tsscmd GetPath pathKey'
```

Element	Description
pathKey	Specifies one of these values: ▪ SOURCE_PATH to get the root path of the test script source from which the currently executing test script was selected. On an agent, this is the root of the destination to which files are copied from the local computer. ▪ ATTACHED_LOG_FILE_PATH to get the root of files attached to the log.

Return Value

On success, this command returns the root of the currently executing test script or of the files attached to the log. On failure, it returns nothing: call `ErrorDetail`for information.

Comments

The root path returned by this command might be the exact location where an asset is stored, but it need not be. For example, in the fully-qualified pathname C:\Datastore\TestScripts, C: might be the root path and Datastore\TestScripts a pathname relative to the root path.

For test scripts run from TestManager, the returned root path is a value in shared memory for the current virtual tester at the time of the call. For test scripts run stand-alone (outside TestManager), the returned root path is a value set by `Context`.

Example

This example returns the root path of the source from which the currently executing test script was selected.

```
scriptPath = 'tsscmd GetPath SOURCE_PATH'
```

See Also

`Context`,`UniqueString`

GetScriptOption

Gets the value of a test script playback option.

Syntax

```
optVal='tsscmd GetScriptOption optionName'
```

Element	Description
optionName	The name of the script option whose value is returned.

Return Value

On success, this command returns the value of the specified script option, or NULL if the value specified is not used by the execution adapter. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

TestManager users can set the values of test script playback options. These may be options specifically supported by a Test Script Execution Adapter (TSEA), or arbitrarily named user-defined options. The common way to use test script options in a test script is to query an option's value with this call and branch according to its returned value.

The Command Line adapter supports these options: `_TM_CMDLINE_WORKING_DIR`, `_TM_CMDLINE_LEFT_PARAM_DELIM`, `_TM_CMDLINE_RIGHT_PARAM_DELIM`, `_TM_CMDLINE_ESCAPE_CHAR`, `_TM_SHELLCMD_ARGS`, `_TM_PASSWORD`, `_TM_USERNAME`, `_TM_PROJECT_PATH`, `_TM_PROJECT_NAME`. These options have initial values that can be changed from TestManager.

Example

This example gets the current working directory.

```
optVal = 'tssscmd GetScriptOption _TM_CMDLINE_WORKING_DIR'
```

GetTestCaseConfigurationAttribute

Gets the value of the specified test case configuration attribute.

Syntax

```
config = 'tsscmd GetTestCaseConfigurationAttribute [-single]
   name'
```

Element	Description
name	Specifies the name of the configuration attribute to be returned.

Return Value

On success, this command returns the value of the specified test case configuration attribute. It exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

You create and maintain test case configuration attributes from TestManager. This command returns the value of the specified attribute for the current test case.

The test case configuration attribute value can be obtained in either of two formats:

- Without the `-single` option, three result lines are returned for each row with the configuration `name` appearing on the first, the `operator` on the second, and the configuration `value` on the third.
- With the `-single` option, each row is returned on one result line containing a `name operator value` triplet.

Example

This example returns the value of the configuration attribute `Operating System`.

```
OSVal = 'tsscmd GetTestCaseConfigurationAttribute "Operating System"'
```

See Also

`GetTestCaseConfigurationAttributeList`

GetTestCaseConfigurationAttributeList

Gets the list of test case configuration attributes and their values.

Syntax

```
config = 'tsscmd GetTestCaseConfigurationAttributeList
   [-single]'
```

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

You create and maintain test case configuration attributes from TestManager. This command returns the current settings for the current test case.

The test case configuration attribute value can be obtained in either of two formats:

- Without the `-single` option, three result lines are returned for each row with the configuration `name` appearing on the first, the `operator` on the second, and the configuration `value` on the third.
- With the `-single` option, each row is returned on one result line containing a `name operator value` triplet.

Example

This example returns the current test case configuration attribute list.

```
config = 'tsscmd GetTestCaseConfigurationAttributeList'
```

See Also

`GetTestCaseConfigurationAttribute`

GetTestCaseConfigurationName

Gets the name of the configuration (if any) associated with the current test case.

Syntax

```
config='tsscmd GetTestCaseConfigurationName'
```

Return Value

On success, this command returns the name of the configuration associated with the test case in use. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

A test case specifies the pass criteria for something that needs to be tested. A configured test case is one that TestManager can execute and resolve as pass or fail.

Example

This example retrieves the name of a test case configuration.

```
tcConfig = 'tsscmd GetTestCaseConfigurationName'
```

GetTestCaseName

Gets the name of the test case in use.

Syntax

```
testcase='tsscmd GetTestCaseName'
```

Return Value

On success, this command returns the name of the current test case. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.

- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

Created from TestManager, a test case specifies the pass criteria for something that needs to be tested.

Example

This example stores the name of the test case in use in `tcName`.

```
tcName = 'tsscmd GetTestCaseName'
```

GetTestToolOption

Gets the value of a test tool execution option.

Syntax

```
optVal='tsscmd GetTestToolOption optionName'
```

Element	Description
optionName	The name of the test tool execution option whose value is returned.

Return Value

On success, this command returns the value of the specified test tool execution option. On failure, it returns nothing: call `ErrorDetail` for information.

Comments

If you develop adapters for a new test script type that support options, you can use this command to get the value of a specified option.

Example

This example returns the value of an option called `persist`.

```
optval = 'tsscmd GetTestToolOption "persist"'
```

JavaApplicationStart

Starts a Java application.

Syntax

```
handle = 'tsscmd JavaApplicationStart [-workdir workingDir]
   [-classpath classPath] [-jvm JVM] [-jvmoptions JVMOptions]
   app'
```

Element	Description
app	The pathname of the application to be started, which can include options and arguments. The file suffix can be omitted.
workingDir	The directory in which to start the application.
classPath	The Java CLASSPATH. The specified value replaces the current CLASSPATH.
JVM	The pathname of Java Virtual Machine. If not specified, java.exe is used on Windows machines and java on UNIX agent platforms.
JVMOptions	Any valid JVM options may be specified.

Return Value

On success, this command returns a handle for the started application. It exits with one of the following values.

- 0 – Success.
- 4 – Server connection failure.
- 5 – The application pathname, classpath, or working directory is invalid.

Example

This example starts application myJavaApp.

```
myAppHandle = 'tsscmd JavaApplicationStart myApp'
```

See Also

ApplicationPid, ApplicationStart, ApplicationWait

NegExp

Gets the next negative exponentially distributed random number with the specified mean.

Syntax

```
nnext='tsscmd NegExp mean'
```

Element	Description
mean	The mean value for the distribution.

Return Value

This command returns the next negative exponentially distributed random number with the specified mean, or –1 if there is an error. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

The behavior of the random number generator routines is affected by the settings of the **Seed** and **Seed Flags** options in a TestManager suite. By default, TestManager sets unique seeds for each virtual tester, so that each has a different random number sequence.

Example

This example seeds the generator and gets a random number with a mean of 10.

```
tsscmd SeedRand 10
next = 'tsscmd NegExp 10'
```

See Also

Rand, SeedRand, Uniform

Rand

Gets the next random number.

Syntax

```
next=‘tsscmd Rand‘
```

Return Value

This command returns the next random number in the range 0 to 32767, or –1 if there is an error. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

The behavior of the random number generator routines is affected by the settings of the **Seed** and **Seed Flags** options in a TestManager suite. By default, TestManager sets unique seeds for each virtual tester, so that each has a different random number sequence.

Example

This example gets the next random number.

```
next = ‘tsscmd Rand‘
```

See Also

```
SeedRand, NegExp, Uniform
```

SeedRand

Seeds the random number generator.

Syntax

```
tsscmd SeedRand seed
```

Element	Description
seed	The base integer.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

The behavior of the random number generator routines is affected by the settings of the **Seed** and **Seed Flags** options in a TestManager suite. By default, TestManager sets unique seeds for each virtual tester, so that each has a different random number sequence.

`SeedRand` uses the argument *`seed`* as a seed for a new sequence of random numbers to be returned by subsequent calls to the `Rand` routine. If `SeedRand` is then called with the same seed value, the sequence of random numbers is repeated. If `Rand` is called before any calls are made to `SeedRand`, the same sequence is generated as when `SeedRand` is first called with a seed value of 1.

Example

This example seeds the random number generator with the number 10:

```
tsscmd SeedRand 10
```

See Also

`Rand, NegExp, Uniform`

ePrint

Prints a message to the virtual tester's error file.

Syntax

```
tsscmd ePrint message
```

Element	Description
message	The string to print.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Example

This example prints to the error file the message `Login failed`. The quotes are optional.

```
tsscmd ePrint "Login failed"
```

See Also

`Print`

Print

Prints a message to the virtual tester's output file.

Syntax

```
tsscmd Print message
```

Element	Description
message	The string to print.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.

- 8 – Pending abort resulting from a user request to stop a suite run.

Example

This example prints the message `Login successful`. The quotes are optional.

```
tsscmd Print "Login successful"
```

See Also

`ePrint`

Uniform

Gets the next uniformly distributed random number.

Syntax

unext=`tsscmd` **`Uniform`** *`low high`*

Element	Description
low	The low end of the range.
high	The high end of the range.

Return Value

This command returns the next uniformly distributed random number in the specified range, or –1 if there is an error. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

The behavior of the random number generator routines is affected by the settings of the **Seed** and **Seed Flags** options in a TestManager suite. By default, TestManager sets unique seeds for each virtual tester, so that each has a different random number sequence.

If the error return value –1 is a legitimate value for the specified range, then `TSSErrorDetail` exits with value 0.

Example

This example gets the next uniformly distributed random number between –10 and 10.

```
next = 'tsscmd Uniform -10 10'
```

See Also

`Rand, SeedRand, NegExp`

UniqueString

Returns a unique text string.

Syntax

str ='tsscmd **UniqueString**'

Return Value

On success, this command returns a string guaranteed to be unique in the current test script or suite run. On failure, it returns NULL: call `ErrorDetail`for information.

Comments

You can use this command to construct the name for a unique asset, such as a test script source file.

Example

This example returns a unique text string.

```
str = 'tsscmd UniqueString'
```

Monitor Commands

When a suite of test cases or test scripts is played back, TestManager monitors execution progress and provides a number of monitoring options. The monitoring commands support the TestManager monitoring options.

Summary

The following table lists the monitoring commands.

Command	Description
`Display`	Sets a message to be displayed by the monitor.
`PositionGet`	Gets the script source file name or line number position.
`PositionSet`	Sets the script source file name or line number position.
`ReportCommandStatus`	Gets the runtime status of a command.
`RunStateGet`	Gets the run state.
`RunStateSet`	Sets the run state.

Display

Sets a message to be displayed by the monitor.

Syntax

`tsscmd` **`Display`** *`message`*

Element	Description
`message`	The message to be displayed by the progress monitor.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 1 – The TSS server is running proxy.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

This message is displayed until overwritten by another call to `Display`.

Example

This example sets the monitor display to `Beginning transaction`. The quotes are optional.

```
tsscmd Display "Beginning transaction"
```

PositionGet

Gets the test script file name or line number position.

Syntax

```
LineAndFile=`tsscmd PositionGet`
```

Return Value

On success, this command returns the name of the source file in use and the current line position. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

TestManager monitoring options include Script View, causing test script lines to be displayed as they are executed. `PositionSet` and `PositionGet` partially support this monitoring option for TSS scripts: if line numbers are reported, they are displayed during playback but not the contents of the lines.

The line number returned by this function is the most recent value that was set by `PositionSet`. A return value of 0 for line number indicates that line numbers are not being maintained.

Example

This example gets the name of the current script file and the number of the line to be accessed next.

```
LineAndFile = `tsscmd PositionGet`
```

See Also

`PositionSet`

PositionSet

Sets the test script file name or line number position.

Syntax

`tsscmd` **`PositionSet`** `[-source` *`srcfile`*`]` *`lineno`*

Element	Description
`srcFile`	The name of the test script, or NULL for the current test script.
`lineNumber`	The number of the line in *`srcFile`* to set the cursor to, or 0 for the current line.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

TestManager monitoring options include Script View, causing test script lines to be displayed as they are executed. `PositionSet` and `PositionGet` partially support this monitoring option for TSS scripts: if line numbers are reported, they are displayed during playback but not the contents of the lines.

Example

This example sets access to the beginning of test script `checkLogin`.

```
tsscmd PositionSet -s checkLogin 0
```

See Also

`PositionSet`

ReportCommandStatus

Reports the runtime status of a command.

Syntax

```
tsscmd ReportCommandStatus status
```

Element	Description
status	The status of a command. Can be one of the following: ▪ FAIL ▪ PASS ▪ WARN ▪ INFO

Return Value

This command exits with one of the following results:

- 0 – Success.
- 1 – The TSS server is running proxy.
- 4 – Server connection failure.
- 5 – The entered *status* is invalid.

- 8 – Pending abort resulting from a user request to stop a suite run.

Example

This example reports a failure command status.

```
tsscmd ReportCommandStatus FAIL
```

RunStateGet

Gets the run state.

Syntax

```
state=`tsscmd RunStateGet`
```

Return Value

On success, this command returns one of the run state values listed in the run state table starting on page 73. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

This call is useful for storing the current run state so you can change the state and then subsequently do a reset to the original run state.

Example

This example gets the current run state.

```
orig = `tsscmd RunStateGet`
```

See Also

RunStateSet

RunStateSet

Sets the run state.

Syntax

```
tsscmd RunStateSet state
```

Element	Description
state	The run state to set. Enter one of the run state values listed in the run state table starting on page 73.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 5 – Invalid run state.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

TestManager includes the option to monitor script progress individually for different virtual testers. The run states are the mechanism used by test scripts to communicate their progress to TestManager. Run states can also be logged and can contribute to performance analysis reports.

The following table lists the TestManager run states.

Run State	Meaning
BIND	iiop_bind in progress
BUTTON	X button action
CLEANUP	cleaning up
CPUDLY	cpu delay
DELAY	user-requested delay
DSPLYRESP	displaying response

Run State	Meaning
EXITED	exited
EXITSQABASIC	exited SQABasic code
EXTERN_C	executing external C code
FIND	find_text find_point
GETTASK	waiting for task assignment
HTTPCONN	waiting for http connection
HTTPDISC	waiting for http disconnect
IIOP_INVOKE	iiop_invoke in progress
INCL	mask including above basic states
INIT	doing startup initialization
INITTASK	initializing task
ITDLY	intertask delay
MOTION	X motion
PMATCH	matching response (precv)
RECV_DELAY	line_speed delay in recv
SATEXEC	executing satellite script
SEND	httpsocket send
SEND_DELAY	line_speed delay in send
SHVBLCK	blocked from shv access
SHVREAD	V_VP: reading shared variable
SHVWAIT	user requested shv wait
SOCKCONN	waiting for socket connection
SOCKDISC	waiting for socket disconnect
SQABASIC_CODE	running SQABasic code
SQLCONN	waiting for SQL client connection
SQLDISC	waiting for SQL client disconnect
SQLEXEC	executing SQL statements

Run State	Meaning
STARTAPP	SQABasic: starting app
SUSPENDED	suspended
TEST	test case, emulate
THINK	thinking
TRN_PACING	transactor pacing delay
TUXEDO	Tuxedo execution
TYPE	typing
UNDEF	user's micro_state is undefined
USERCODE	SQAVu user code
WAITOBJ	SQABasic: waiting for object
WAITRESP	waiting for response
WATCH	interactive -W watch record
XCLNTCONN	waiting for http connection
XCLNTCONN	waiting for socket connection
XCLNTCONN	waiting for SQL client connection
XCLNTCONN	waiting for X client connection
XCLNTDISC	waiting for http disconnect
XCLNTDISC	waiting for socket disconnect
XCLNTDISC	waiting for SQL client disconnect
XCLNTDISC	waiting for X client disconnect
XMOVEWIN	X move window
XQUERY	X query function
XSYNC	X sync state during X query
XWINCMP	xwindow_diff comparing windows
XWINDUMP	xwindow_diff dumping window
N_INCL	number of above states

Example

This example sets the run state to WAITRESP.

```
tsscmd RunStateSet WAITRESP
```

See Also

RunStateGet

Synchronization Commands

Use the synchronization commands to synchronize virtual testers during script playback. You can insert synchronization points and wait periods, and you can manage variables shared among virtual testers.

Summary

The following table lists the synchronization commands.

Command	Description
SharedVarAssign	Performs a shared variable assignment operation.
SharedVarEval	Gets the value of a shared variable and operates on the value as specified.
SharedVarWait	Waits for the value of a shared variable to match a specified range.
SyncPoint	Puts a synchronization point in a script.

SharedVarAssign

Performs a shared variable assignment operation.

Syntax

value=tsscmd **SharedVarAssign** [-quiet] *name value* [*op*]

Element	Description
-quiet	This option suppresses the returned value. If omitted, the statement returns the resulting value of *name* after application of *op value*.
name	The name of the shared variable to operate on.
value	The right-side value of the assignment expression.
op	Assignment operator. Can be one of the following: ▪ assign (default) ▪ add ▪ subtract ▪ multliply ▪ divide ▪ modulo ▪ and ▪ or ▪ xor ▪ shiftleft ▪ shiftright

Return Value

On success, this command retrieves the value of the specified shared variable. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 5 – The entered *name* is not a shared variable.
- 8 – Pending abort resulting from a user request to stop a suite run.

Example

This example adds 5 to the value of the shared variable `lineCounter` and puts the new value of `lineCounter` in `returnval`.

```
returnval = `tsscmd SharedVarAssign lineCounter 5 add`
```

See Also

`SharedVarEval, SharedVarWait`

SharedVarEval

Gets the value of a shared variable and operates on the value as specified.

Syntax

```
value=`tsscmd SharedVarEval name [op`]`
```

Element	Description
name	The name of the shared variable to operate on.
op	Increment/decrement operator for the returned value: Can be one of the following: ▪ none (default) ▪ pre_inc ▪ post_inc ▪ pre_dec ▪ post_dec

Return Value

On success, this command returns the new value of the specified shared variable. The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 5 – The entered *name* is not a shared variable.
- 8 – Pending abort resulting from a user request to stop a suite run.

Example

This example post-decrements the value of shared variable `lineCounter` and stores the result in `val`.

```
val = `tsscmd SharedVarEval lineCounter post_inc`
```

See Also

SharedVarAssign, SharedVarWait

SharedVarWait

Waits for the value of a shared variable to match a specified range.

Syntax

```
returnVal=`tsscmd SharedVarWait [-quiet] [-adjust adjust]
   [-timeout timeout] name min [max]`
```

Element	Description
-quiet	This option suppresses the returned value. If omitted, the statement returns the value of *name* before any possible adjustment.
name	The name of the shared variable to operate on.
min	The low range for the value of *name*.
max	The high range for the value of *name*.
adjust	The value to increment/decrement the named shared variable by once it meets the *min* – *max* range.
timeout	The time-out preference (how long to wait for the condition to be met). Enter one of the following: ▪ A negative number for no time-out. ▪ 0 to return immediately with an exit value of 1 (condition met) or 0 (not met). ▪ The number of milliseconds to wait for the value of *name* to meet the criteria, before timing out with and returning an exit value of 1 (met) or 0 (not met).

Return Value

The command exits with one of the following results:

- 0 – The shared variable did not meet the range during the time-out period.
- 1 – The shared variable met the range during the time-out period.
- 4 – Server connection failure.
- 5 – The entered *name* is not a shared variable.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

This call provides a method of blocking a virtual tester until a user-defined global event occurs.

If virtual testers are blocked on an event using the same shared variable, TestManager guarantees that the virtual testers are unblocked in the same order in which they were blocked.

Although this *alone* does not ensure an exact multiuser timing order in which statements following a `wait` are executed, the additional proper use of the arguments *`min`*, *`max`*, and *`adjust`* allows control over the order in which multiuser operations occur. (UNIX or Windows NT determines the order of the scheduling algorithms. For example, if two virtual testers are unblocked from a wait in a given order, the tester that was unblocked last might be released before the tester that was unblocked first.)

If a shared variable's value is modified, any subsequent attempt to modify this value — other than through `SharedVarWait` — blocks execution until all virtual testers already blocked have had an *opportunity* to unblock. This ensures that events cannot appear and then quickly disappear before a blocked virtual tester is unblocked. For example, if two virtual testers were blocked waiting for *`name`* to equal or exceed *`N`*, and if another virtual tester assigned the value *`N`* to *`name`*, then TestManager guarantees both virtual testers the opportunity to unblock before any other virtual tester is allowed to modify *`name`*.

Offering the *opportunity* for all virtual testers to unblock does not guarantee that all virtual testers actually unblock, because if `SharedVarWait` is called with a nonzero value of *`adjust`* by one or more of the blocked virtual testers, the shared variable value changes during the unblocking script. In the previous example, if the first user to unblock *had* called `SharedVarWait` with a negative *`adjust`* value, the event waited on by the second user would no longer be true after the first user unblocked. With proper choice of *`adjust`* values, you can control the order of events.

Example

This example returns `1` if the shared variable `inProgress` reaches a value between 10 and 20 within 60000 milliseconds of the time of the call. Otherwise, it returns `0`. `svVal` contains the value of `inProgress` at the time of the return, before it is adjusted. (In this case, the adjustment value is 0 so the value of the shared variable is not adjusted.)

```
svVal = SharedVarWait -t 60000 inProgress 10 20
```

See Also

`SharedVarAssign, SharedVarEval`

SyncPoint

Puts a synchronization point in a script.

Syntax

```
tsscmd SyncPoint label
```

Element	Description
label	The name of the synchronization point.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 1 – The TSS server is running proxy.
- 4 – Server connection failure.
- 5 – The synchronization point *label* is invalid.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

A script pauses at a synchronization point until the release criteria specified by the suite have been met. If the criteria are met, the script delays a random time specified in the suite and then resumes execution.

Typically, it is better to insert a synchronization point into a suite from TestManager rather than use the `SyncPoint` call inside a script.

If you insert a synchronization point into a suite, synchronization occurs at the beginning of the script. If you insert a synchronization point into a script with `SyncPoint`, synchronization occurs at the point of insertion. You can insert the command anywhere in the script.

Example

This example creates a sync point named `BlockUntilSaveComplete`.

```
tsscmd SyncPoint BlockUntilSaveComplete
```

Session Commands

This section documents functions that may be required by applications. They are not typically used by test scripts.

A suite can contain multiple test scripts of different types. When TestManager executes a suite, a separate *session* is started for each type of script in the suite. Each session lasts until all scripts of the type have finished executing. Thus, if a suite contains three Visual Basic test scripts and six VU test scripts, two sessions are started and each remains active until all scripts of the respective types finish.

tsscmd statements are executed outside TestManager, by a proxy TSS server process. If TestManager (or **rttsee**) encounters a **tsscmd** statement and no proxy server process is running, one is started. Each **tsscmd** statement connects to this process, and then disconnects after the service completes.

Summary

Applications can use the session commands listed in the following table to manage proxy TSS servers and sessions on behalf of test scripts. commands.

Command	Description
`Context`	Passes context information to a TSS server.
`ServerStart`	Starts a TSS proxy server.
`ServerStop`	Stops a TSS proxy server.

Context

Passes context information to a TSS server.

Syntax

```
tsscmd Context ctx value
```

Element	Description
ctx	The type of context information to pass: Can be one of the following: ▪ workingDir ▪ datapoolDir ▪ timeZero ▪ todZero ▪ logDir ▪ logFile ▪ logData ▪ testScript ▪ style ▪ sourceUID
value	The information of type *ctx* to pass.

Return Value

This commandexits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 5 – The specified *ctx* is invalid.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

This command passes information, such as the log file name, that would be passed through shared memory if the script were executed by TestManager. Where used in a script, it should be used first, before any other **tsscmd** command. Otherwise, inconsistent results can occur.

Example

This example passes a working directory to the current proxy TSS server.

```
tsscmd Context workingDir "C:\temp"
```

ServerStart

Starts a TSS proxy server.

Syntax

```
p=‘tsscmd ServerStart [port]‘
```

Element	Description
port	The listening port for the TSS server. If omitted (recommended), the system chooses the port and returns its number to *p*.

Return Value

This command exits with one of the following results:

- 0 – Success.
- 1 – A TSS server was already listening on *port*.
- 4 – Start failure. Call `ErrorDetail` for information.
- 6 – A system error occurred. Call `ErrorDetail` for information.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

No TSS server is started if one is already running. A test script that is to be executed by a proxy server and that might be the first to execute should make this call.

Example

This example starts a proxy TSS server on a system-designated port, whose number is returned to *port*.

```
port = ‘tsscmd ServerStart‘
```

See Also

`ServerStop`

ServerStop

Stops a TSS proxy server.

Syntax

```
tsscmd ServerStop port
```

Element	Description
port	The port number that the TSS server to be stopped is listening on.

Return Value

This commandexits with one of the following results:

- 0 – Success.
- 1 – No TSS server was listening on *port*.
- 5 – No proxy TSS server was found or stopped.
- 6 – A system error occurred. Call `ErrorDetail` for information.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

In a test suite with multiple scripts, only the last executed script should make this call.

Example

This example stops a proxy TSS server listening on port 3825.

```
tsscmd ServerStop 3825
```

See Also

`ServerStart`

Advanced Commands

You can use the advanced commands to perform timing calculations, logging operations, and internal variable initialization functions. TestManager performs these operations on behalf of scripts in a safe and efficient manner. Consequently, the functions need not and usually should not be performed by individual test scripts.

Summary

The following table lists the advanced commands.

Command	Description
`InternalVarSet`	Sets the value of an internal variable.
`LogCommand`	Logs a command event.
`ThinkTime`	Calculates a think-time average.

InternalVarSet

Sets the value of an internal variable.

Syntax

`tsscmd` **`InternalVarSet`** *`internVar ivVal`*

Element	Description
`internVar`	The internal variable to operate on. Internal variables and their values are listed in the table starting on page 99.
`ivVal`	The new value for *`internVar`*.

Return Value

The command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.

- 5 – The timer label is invalid, or there is no unlabeled timer to stop.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

The values of some internal variables affect think-time calculations and the contents of log events. Setting a value incorrectly could cause serious misbehavior in a script.

Example

This example sets `cmdcnt` to 0.

```
tsscmd InternalVarSet cmdcnt 0
```

See Also

`InternalVarGet`

LogCommand

Logs a command event.

Syntax

```
tsscmd LogCommand [-desc description] [-start starttime] [-end
   endtime] name label result logdata [property=value ...]
```

Element	Description
description	Contains the string to be displayed in the event of failure.
starttime	An integer indicating a time stamp. If omitted or specified as 0, the logged time stamp is the later of the values contained in internal variables `fcs_ts` and `fcr_ts`.
endtime	An integer indicating a time stamp. If omitted or specified as 0, the time set by `CommandEnd` is logged.
name	The command name.
label	The event label.

Element	Description
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ NONE (default: no notification) ▪ PASS ▪ FAIL ▪ WARN ▪ STOPPED ▪ INFO ▪ COMPLETED ▪ UNEVALUATED
logdata	Text to be logged describing the ended command.
property=value	Specifies one or more property-value pairs

Return Value

This command exits with one of the following results:

- 0 – Success.
- 4 – Server connection failure.
- 8 – Pending abort resulting from a user request to stop a suite run.

Comments

The value of `cmdcnt` is logged with the event.

The command name and label entered with `CommandStart` are logged, and the run state is restored to the value that existed prior to the `CommandStart` call.

An event and any data associated with it are logged only if the specified *result* preference matches associated settings in the `LogData_control` (page 92) or `LogEvent_control` (page 92) environment variables. Alternatively, the logging preference may be set with the `Log_level` (page 93) and `Record_level` (page 94) environment variables. The STOPPED, COMPLETED, and UNEVALUATED preferences are intended for internal use.

Example

This example logs a message for a login script.

```
tsscmd LogCommand -d "Command timer failed" Login initTimer PASS
```

See Also

`CommandStart`, `CommandEnd`

ThinkTime

Calculates a think-time average.

Syntax

```
thinkTime = `tsscmd ThinkTime [thinkAverage]`
```

Element	Description
thinkAverage	If specified as 0, the number of milliseconds stored in the `ThinkAvg` environment variable is entered. Otherwise, the value specified overrides `ThinkAvg`.

Return Value

On success, this command returns a calculated think-time average. An exit value of 1 indicates an error. Call `ErrorDetail` for more information.

Comments

This call calculates and returns a think time using the same algorithm as `Think`. But unlike `Think`, this call inserts no pause into a script.

This function could be useful in a situation where a test script calls another program that, as a matter of policy, does not allow a calling program to set a delay in execution. In this case, the called program would use `ThinkTime` to recalculate the delay requested by `Think` before deciding whether to honor the request.

Example

This example calculates a pause based on a think-time average of 5000 milliseconds.

```
ctime = ‘tsscmd GetTime‘
tsscmd InternalVarSet fcs_ts ctime
tsscmd InternalVarSet lcs_ts ctime
tsscmd InternalVarSet fcr_ts ctime
tsscmd InternalVarSet fcr_ts ctime
pause = ‘tsscmd ThinkTime 5000‘
```

See Also

Think

Environment and Internal Variable Arguments

A

This appendix documents the predefined constants that can be used as arguments with the statements you use to set environment variables and to retrieve internal variable values, and provides an example illustrating how to manipulate environment variables.

Arguments of EnvironmentOp

The following table describes the valid values of the first argument (*envVar*) of `EnvironmentOp`. Note the following about `LogData_control` and `LogEvent_control`:

- They correspond to the check boxes in the TestManager TSS Environment Variables dialog box. Use this dialog box to set logging and reporting options at the suite rather than the script level.
- They are more flexible alternatives to `Log_level` and `Report_level`.

Name	Type/Values/(default)	Contains
`Delay_dly_scale`	integer 0–2000000000 percent (100)	The scaling factor applied globally to all timing delays. A value of 100%, which is the default, means no change. A value of 50% means one-half the delay, which is twice as fast as the original; 200% means twice the delay, which is half as fast. A value of zero means no delay.

Name	Type/Values/(default)	Contains
LogData_control	NONE, PASS, FAIL, WARNING, STOPPED, INFORMATIONAL, COMPLETED, UNEVALUATED ANYRESULT	Flags indicating the level of detail to log. Specify one or more. These result flags (except the last, which specifies everything) correspond to flags entered with the LogEvent, LogMessage, TestCaseResult, CommandEnd, and LogCommand statements. For example, specifying FAIL selects everything logged by statements that specified flag FAIL.
LogEvent_controlL	NONE, PASS, FAIL, WARNING, STOPPED, INFORMATIONAL, COMPLETED, UNEVALUATED, TIMERS, COMMANDS, ENVIRON, STUBS, TSSERROR, TSSPROXYERROR ANYRESULT	Flags indicating the level of detail to log for reports. Specify one or more. The first nine result flags (NONE through UNEVALUATED) correspond to flags specified with the LogEvent, LogMessage, TestCaseResult, CommandEnd, and LogCommand statements. The other flags (TIMERS through TSSPROXYERROR) indicate the event objects. For example, FAIL plus COMMANDS selects for reporting all commands that recorded a failed result. ANYRESULTS selects everything.

Name	Type/Values/(default)	Contains
`Log_level`	string "OFF" ("TIMEOUT") "UNEXPECTED" "ERROR" "ALL"	The level of detail to log: ▪ `OFF` – Log nothing. ▪ `TIMEOUT` – Log emulation command time-outs. ▪ `UNEXPECTED` – Log time-outs and unexpected responses from emulation commands. ▪ `ERROR` – Log all emulation commands that set `error` to a nonzero value. Log entries include `error` and `error_text`. ▪ `ALL` – Log everything: emulation command types and IDs, script IDs, source files, and line numbers.

Name	Type/Values/(default)	Contains
Record_level	"MINIMAL" "TIMER" "FAILURE" ("COMMAND") "ALL"	The level of detail to log for reporting: ▪ MINIMAL – Record only items necessary for reports to run. Use this value when you do not want user activity to be reported. ▪ TIMER – MINIMAL plus start_time and stop_time emulation commands. Reports do not contain response times for each emulation command, emulation command failure does not appear, and the result file for each virtual tester is small. Use this setting if you are not concerned with the response times or pass/fail status of individual emulation commands. ▪ FAILURE – TIMER plus emulation command failures and some environment variable changes. Use this setting if you want the advantages of a small result file but to show also that no emulation command failed. ▪ COMMAND – FAILURE plus emulation command successes and some environment variable changes. ▪ ALL – COMMAND plus all environment variable changes. Complete recording.

Name	Type/Values/(default)	Contains
Suspend_check	string ("ON") "OFF"	Controls whether you can suspend a virtual tester from a Monitor view: ▪ ON – A suspend request is checked before beginning the think time interval by each send emulation command. ▪ OFF – Disable suspend checking.
Think_avg	integer 0–2000000000 ms (5000)	The average think-time delay (the amount of time that, on average, a user delays before performing an action).
Think_cpu_dly_scale	integer 0–2000000000 ms (100)	The scaling factor applied globally to CPU (processing time) delays. Used instead of Think_dly_scale if Think_avg is less than Think_cpu_threshold. Delay scaling is performed before truncation (if any) by Think_max.
Think_cpu_threshold	integer 0–2000000000 ms (0)	The threshold value used to distinguish CPU delays from think-time delays.

Name	Type/Values/(default)	Contains
Think_def	string "FS" "LS" "FR" ("LR") "FC" "LC"	The starting point of the think-time interval: ▪ FS – the submission time of the previous send emulation command ▪ LS – the completion time of the previous send emulation command ▪ FR – the time the first data of the previous receive emulation command was received ▪ LR – the time the last data of the previous receive emulation command was received, or LS if there was no intervening receive emulation command ▪ FC – the submission time of the previous connect emulation command (uses the fc_ts internal variable) ▪ LC – the completion time of the previous connect emulation command (uses the lc_ts internal variable)
Think_dist	string ("CONSTANT") "UNIFORM" "NEGEXP"	The think-time distribution: ▪ CONSTANT – sets a constant distribution equal to Think_avg ▪ UNIFORM – sets a random think-time interval distributed uniformly in the range: [Think_avg - Think_sd, Think_avg + Think_sd] ▪ NEGEXP – sets a random think-time interval approximating a bell curve with Think_avg equal to standard deviation

Name	Type/Values/(default)	Contains
Think_dly_scale	integer 0 – 2000000000 ms (100)	The scaling factor applied globally to think-time delays. Used instead of Think_cpu_dly_scale if Think_avg is greater than Think_cpu_threshold. Delay scaling is performed before truncation (if any) by Think_max.
Think_max	integer 0–2000000000 ms (2000000000)	A maximum threshold for think times that replaces any larger setting.
Think_sd	integer 0–2000000000 ms (0)	Where Think_dist is set to UNIFORM, specifies the think-time standard deviation.

The following table describes the valid values of the second argument (*envOp*) of EnvironmentOp.

Operation	Description
eval	Operate on the value at the top of the variable's stack.
pop	Remove the variable value at the top of the stack.
push	Push a value to the top of a variable's stack.
reset	Set the value of a variable to the default and discard any other values in the stack.
restore	Set the saved value to the current value.
save	Save the value of a variable.
set	Set a variable to the specified value.

Example: Manipulating Environment Variables

This example illustrates how to manipulate environment variables.

```
$ev = 'tsscmd environmentop think_dist eval';
errorexit() if $?;
```

```
chomp($ev);
print "At start, value is '$ev'\n";

system("tsscmd environmentop think_dist push NEGEXP") == 0
    or errorexit();
$ev = `tsscmd environmentop think_dist eval`;
errorexit() if $?;
chomp($ev);
print "After push, value is '$ev'\n";

system("tsscmd environmentop think_dist pop") == 0
    or errorexit();
$ev = `tsscmd environmentop think_dist eval`;
errorexit() if $?;
chomp($ev);
print "After pop, value is '$ev'\n";

system("tsscmd environmentop think_dist set NEGEXP") == 0
    or errorexit();
$ev = `tsscmd environmentop think_dist eval`;
errorexit() if $?;
chomp($ev);
print "After set, value is '$ev'\n";

system("tsscmd environmentop think_dist save") == 0
    or errorexit();
system("tsscmd environmentop think_dist reset") == 0
    or errorexit();
$ev = `tsscmd environmentop think_dist eval`;
errorexit() if $?;
chomp($ev);
print "After save and reset, value is '$ev'\n";

system("tsscmd environmentop think_dist restore") == 0
    or errorexit();
$ev = `tsscmd environmentop think_dist eval`;
errorexit() if $?;
chomp($ev);
print "After restore, value is '$ev'\n";


sub errorexit {
    my $msg, $r;

    $msg = `tsscmd errordetail`;
    $r = $?>>8;
    chomp($msg);
    die "tsscmd call failed, code $r: $msg\n";
}
```

Arguments of InternalVarGet

The following table lists the internal variables that can be entered with the *internVar* argument.

Variable	Contains
alltext	Response text up to the value of Max_nrecv_saved. The same as response.
cmd_id	The ID of the most recent emulation command.
cmdcnt	A running count of the number of emulation commands the script has executed.
col	The current column position (1-based) of the cursor (ASCII screen emulation variable).
column_headers	The two-line column header if Column_headers is ON; otherwise, empty.
command	The text of the most recent emulation command.
cursor_id	The last cursor declared by sqldeclare_cursor or opened by sqlopen_cursor.
error	The status of the last emulation command. Most values for error are supplied by the server.
error_text	The full text of the error from the last emulation command. If error is 0, error_text returns nothing. For a SQL database or TUXEDO error, the text is provided by the server.
error_type	If you are emulating a TUXEDO session and error is nonzero, error_type contains one of the following values: 0 (no error) 1 VU/TUX Usage Error 2 TUXEDO System/T Error 3 TUXEDO FML Error 4 TUXEDO FML32 Error 5 Application under test Error 6 Internal Error If you are emulating an IIOP session and error is nonzero, error_type contains one of the following values: 0 (no error) 1 IIOP_EXCEPTION_SYSTEM 2 IIOP_EXCEPTION_USER 3 IIOP_ERROR

Variable	Contains
fc_ts	The "first connect" time stamp for http_request and sock_connect.
fr_ts	The time stamp of the first received data of sqlnrecv, http_nrecv, http_recv, http_header_recv, sock_nrecv, or sock_recv. For sqlexec and sqlprepare, fr_ts is set to the time the SQL database server responded to the SQL statement.
fs_ts	The time the SQL statement was submitted to the server by sqlexec or sqlprepare, or the time when the first data was submitted to the server by http_request or sock_send.
host	The host name of the computer on which the script is running.
lc_ts	The "last connect" time stamp for http_request and sock_connect.
lineno	The line number in source_file of the previously executed emulation command.
lr_ts	The time stamp of the last received data for sqlnrecv, http_nrecv, http_recv, http_header_recv, sock_nrecv, or sock_recv. For sqlexec and sqlprepare, lr_ts is set to the time the SQL database server responded to the SQL statement.
ls_ts	The time the SQL statement was submitted to the server by sqlexec or sqlprepare, or the time the last data was submitted to the server by http_request or sock_send.
mcommand	The actual (mapped) sequence of characters submitted to the application under test by the most recent send or msend command. For send commands, mcommand is always equivalent to command.
ncnull	The number of null characters in an application response examined by the previous receive command in attempting to match this response.
ncols	The number of columns in the current screen (ASCII screen emulation variable).
ncrecv	The total number of nonnull characters from an application response examined by the previous receive command in attempting to match this response.
ncxmit	The total number of characters transmitted to the application by the previous send or msend command.

Variable	Contains
nkxmit	The total number of "keystrokes" transmitted to the application by the previous send or msend command. For send commands, nkxmit is always equivalent to ncxmit.
nrecv	The number of rows processed by the last sqlnrecv, or the number of bytes received by the last http_nrecv, http_recv, sock_nrecv, or sock_recv.
nrows	The number of rows in the current screen (ASCII screen emulation variable).
nusers	The number of total virtual testers in the current TestManager session.
nxmit	The total number of characters contained in the SQL statements transmitted to the server in the last sqlexec or sqlprepare command, or the number of bytes transmitted by the last http_request or sock_send.
response	Same as row.
row	The current row position (1-based) of the cursor (ASCII screen emulation variable).
script	The name of the script currently being executed.
source_file	The name of the file that was the source for the portion of the script being executed.
statement_id	The value assigned as the prepared statement ID, which is returned by sqlprepare and sqlalloc_statement.
total_nrecv	The total number of bytes received for all HTTP and socket receive emulation commands issued on a particular connection.
total_rows	Set to the number of rows processed by the SQL statements. If the SQL statements do not affect any rows, total_rows is set to 0. If the SQL statements return row results, total_rows is set to 0 by sqlexec, and then incremented by sqlnrecv as the row results are retrieved.
tux_tpurcode	TUXEDO user return code, which mirrors the TUXEDO API global variable tpurcode. It can be set only by the tux_tpcall, tux_tpgetrply, tux_tprecv, and tux_tpsend emulation commands.
uid	The numeric ID of the current virtual tester.
user_group	The name of the user group (from the suite) of the virtual tester running the script.
version	The full version string of TestManager (for example, 7.5.0.1045).

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W