

Rational® TestManager

EXTENSIBILITY REFERENCE

VERSION: 2002.05.00

PART NUMBER: 800-025136-000

IMPORTANT NOTICE

COPYRIGHT

Copyright ©2000-2001, Rational Software Corporation. All rights reserved.

Part Number: 800-025136-000

Version Number: 2002.05.00

PERMITTED USAGE

THIS DOCUMENT CONTAINS PROPRIETARY INFORMATION WHICH IS THE PROPERTY OF RATIONAL SOFTWARE CORPORATION ("RATIONAL") AND IS FURNISHED FOR THE SOLE PURPOSE OF THE OPERATION AND THE MAINTENANCE OF PRODUCTS OF RATIONAL. NO PART OF THIS PUBLICATION IS TO BE USED FOR ANY OTHER PURPOSE, AND IS NOT TO BE REPRODUCED, COPIED, ADAPTED, DISCLOSED, DISTRIBUTED, TRANSMITTED, STORED IN A RETRIEVAL SYSTEM OR TRANSLATED INTO ANY HUMAN OR COMPUTER LANGUAGE, IN ANY FORM, BY ANY MEANS, IN WHOLE OR IN PART, WITHOUT THE PRIOR EXPRESS WRITTEN CONSENT OF RATIONAL.

TRADEMARKS

Rational, Rational Software Corporation, the Rational logo, Rational the e-development company, ClearCase, ClearQuest, Object Testing, Object-Oriented Recording, Objectory, PerformanceStudio, PureCoverage, PureDDTS, PureLink, Purify, Purify'd, Quantify, Rational Apex, Rational CRC, Rational PerformanceArchitect, Rational Rose, Rational Suite, Rational Summit, Rational Unified Process, Rational Visual Test, Requisite, RequisitePro, SiteCheck, SoDA, TestFactory, TestMate, TestStudio, and The Rational Watch are trademarks or registered trademarks of Rational Software Corporation in the United States and in other countries. All other names are used for identification purposes only, and are trademarks or registered trademarks of their respective companies.

Microsoft, the Microsoft logo, the Microsoft Internet Explorer logo, DeveloperStudio, Visual C++, Visual Basic, Windows, the Windows CE logo, the Windows logo, Windows NT, the Windows Start logo, and XENIX are trademarks or registered trademarks of Microsoft Corporation in the United States and other countries.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

FLEXlm and GLOBEtrotter are trademarks or registered trademarks of GLOBEtrotter Software, Inc. Licensee shall not incorporate any GLOBEtrotter software (FLEXlm libraries and utilities) into any product or application the primary purpose of which is software license management.

PATENT

U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,835,701. Additional patents pending.

Purify is licensed under Sun Microsystems, Inc., U.S. Patent No. 5,404,499.

GOVERNMENT RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the applicable Rational Software Corporation license agreement and as provided in DFARS 277.7202-1(a) and 277.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (Oct. 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 227-14, as applicable.

WARRANTY DISCLAIMER

This document and its associated software may be used as stated in the underlying license agreement. Rational Software Corporation expressly disclaims all other warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability or fitness for a particular purpose or arising from a course of dealing, usage, or trade practice.

Contents

Preface	xi
About This Manual	xi
Audience	xi
Other Resources	xi
Contacting Rational Technical Publicationsxii
Contacting Rational Technical Supportxii
 Part 1: Adding Custom Test Script Types	
1 Introduction to Custom Test Script Types	1
About Test Script Types	1
Built-In Test Script Types	1
Adding Test Script Types	2
Adding a Command Line Test Script Type	3
Adding a Custom Test Script Type	9
Component Description and Communication Overview	10
Using the Command Line Execution Engine	12
2 Test Script Execution Adapter API	15
About This API	15
Communication Overview	15
Data Types and C Header Files	17
Summary	17
SessionClose()	18
SessionGetOption()	18
SessionOpen()	19
SessionSetOption()	20
TaskAbort()	20
TaskClose()	21
TaskCreate()	21
TaskExecute()	22
TaskGetOption()	22
TaskSetOption()	23
TSELError()	24

3 Test Script Services	25
About Test Script Services	25
Datapool Services	25
Summary	26
TSSDatapoolClose()	27
TSSDatapoolColumnCount()	27
TSSDatapoolColumnName()	28
TSSDatapoolFetch()	29
TSSDatapoolOpen()	30
TSSDatapoolRewind()	33
TSSDatapoolRowCount()	34
TSSDatapoolSearch()	35
TSSDatapoolSeek()	36
TSSDatapoolValue()	37
Logging Services	38
Summary	39
TSSLogEvent()	39
TSSLogMessage()	41
TSSLogTestCaseResult()	42
Measurement Services	44
Summary	44
TSSCommandEnd()	45
TSSCommandStart()	47
TSSEnvironmentOp()	48
TSSGetTime()	50
TSSInternalVarGet()	51
TSSThink()	53
TSSTimerStart()	54
TSSTimerStop()	55
Utility Services	57
Summary	57
TSSApplicationPid()	58
TSSApplicationStart()	59
TSSApplicationWait()	60
TSSDelay()	61
TSSErrorDetail()	62
TSSGetComputerConfigurationAttributeList()	63
TSSGetComputerConfigurationAttributeValue()	64

TSSGetPath()	65
TSSGetScriptOption()	66
TSSGetTestCaseConfigurationAttribute()	67
TSSGetTestCaseConfigurationAttributeList()	68
TSSGetTestCaseConfigurationName()	69
TSSGetTestCaseName()	70
TSSGetTestToolOption()	70
TSSJavaApplicationStart()	71
TSSNegExp()	72
TSSRand()	73
TSSSeedRand()	74
TSSePrint()	75
TSSPrint()	76
TSSUniform()	76
TSSUniqueString()	77
Monitor Services	78
Summary	78
TSSDisplay()	79
TSSPositionGet()	79
TSSPositionSet()	81
TSSReportCommandStatus()	82
TSSRunStateGet()	83
TSSRunStateSet()	83
Synchronization Services	87
Summary	87
TSSSharedVarAssign()	87
TSSSharedVarEval()	89
TSSSharedVarWait()	90
TSSSyncPoint()	93
Session Services	94
Summary	94
TSSConnect()	95
TSSContext()	96
TSSDisconnect()	98
TSSServerStart()	98
TSSServerStop()	99
TSSShutdown()	100

Advanced Services	101
Summary	101
TSSInternalVarSet()	101
TSSLogCommand()	102
TSSThinkTime()	104
4 Test Script Console Adapter API	107
About the Test Script Console Adapter	107
TSCA Functionality	107
Built-In and Custom Test Script Types	108
The TSCA Function Calls	109
Functional Groupings of TSCA Functions	109
Required and Optional Functionality	110
Mapping of User Actions to TSCA Function Calls	111
Building a Custom Test Script Console Adapter	116
Prerequisite Skills	116
Building a TSCA: Workflow and Implementation Issues	116
Making a Connection	116
Accessing the Data	117
Integration with Source Control	119
Displaying Properties	119
Supporting User Configuration of the Test Script Source	119
Filtering	120
Custom Action Support	120
Registering the TSCA DLL with TestManager	121
TSCA Function Reference	121
TTAddToSourceControl()	122
TTCheckIn()	124
TTCheckOut()	125
TTClearFilter()	127
TTConnect()	128
TTDisconnect()	132
TTEdit()	133
TTExecuteNodeAction()	135
TTExecuteSourceAction()	136
TTGetChildren()	138
TTGetConfiguration()	140
TTGetFilterEx()	142
TTGetIcon()	144
TTGetIsFunctionSupported()	145

TTGetName()	147
TTGetNode()	149
TTGetNodeActions()	151
TTGetRoots()	152
TTGetSourceActions()	153
TTGetSourceControlStatus()	155
TTGetSourceIcon()	157
TTGetTestToolOptions()	158
TTGetTypeIcon()	161
TTNew()	162
TTSelect()	164
TTSetConfiguration()	166
TTSetFilterEx()	168
TTShowProperties()	170
TTUndoCheckout()	172

Part 2: Adding Custom Test Input Types

5 Introduction to the Test Input Adapter API	177
About Test Input Adapters	177
TIA Functionality	177
Built-In and Custom TIAs	178
The TIA Function Calls	179
Functional Groupings of TIA Functions	179
Mapping of User Actions to TIA Function Calls	181
Building a Custom Test Input Adapter	185
Prerequisite Skills	185
Building a TIA: Workflow and Implementation Issues	185
Making a Connection	186
Accessing the Data	187
Supporting Impact Analysis	188
Displaying Properties	189
Supporting User Configuration of Test Input Data	189
Filtering	190
Custom Action Support	190
Registering a New Test Input Adapter	191

6 Test Input Adapter Reference	193
Summary of TIA Functions	193
Using the Type Node Structure	195
Note on Memory Allocation	196
TIConnect()	196
TIConnectEx()	199
TIDisconnect()	201
TIExecuteNodeAction()	203
TIExecuteSourceAction()	204
TIGetChildren()	206
TIGetConfiguration()	209
TIGetFilterEx()	211
TIGetIsChild()	213
TIGetIsFunctionSupported()	214
TIGetIsModified()	216
TIGetIsModifiedSince()	217
TIGetIsNode()	218
TIGetIsParent()	220
TIGetIsValidSource()	221
TIGetModified()	222
TIGetModifiedSince()	223
TIGetName()	226
TIGetNeedsValidation()	227
TIGetNode()	228
TIGetNodeActions()	230
TIGetParent()	232
TIGetRoots()	234
TIGetSourceActions()	237
TIGetSourceIcon()	238
TIGetType()	240
TIGetTypeIcon()	241
TIGetTypes()	243
TISetConfiguration()	245
TISetFilter()	246

TISetFilterEx()	248
TISetValidationFilter()	250
TIShowProperties()	251
TIShowSelectDialog()	253
Index	257

Preface

About This Manual

This manual describes the APIs that you use to extend the capabilities of Rational TestManager. The manual is divided into two parts:

- *Part 1: Adding Custom Test Script Types*

Part 1 describes the APIs you use if you want TestManager to manage and run custom test script types in addition to the standard test types that it supports (such as SQABasic, VU, Java, and Visual Basic test scripts as well as shell scripts and manual scripts).

- *Part 2: Adding Custom Test Input Types*

Part 2 describes the APIs you use to introduce new test inputs (such as custom test requirements and other custom test assets) into the TestManager environment.

Audience

This manual is intended for developers who write TestManager adapters that interface with TestManager's C execution engine.

Other Resources

- This product contains online documentation. To access it, click **TestManager Extensibility** 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>.

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 Technical Support

If you have questions about installing, using, or maintaining this product, contact Rational Technical 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 Technical 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)

Part 1: Adding Custom Test Script Types

Introduction to Custom Test Script Types

1

About Test Script Types

Part 1 of this manual explains how to customize Rational® TestManager so that it can execute a test script written in a new language. Part 1 is organized as described in the following table.

Section	Purpose
<i>Built-In Test Script Types</i> on page 1	Explains the role of test script types in execution and describes built-in test script types.
<i>Adding Test Script Types</i> on page 2	Describes the two ways to extend TestManager to support a new test script type or testing environment.
<i>Adding a Command Line Test Script Type</i> on page 3	Gives the step-by-step procedure you follow to create a test script type that uses Rational's Command Line test script adapter.
<i>Adding a Custom Test Script Type</i> on page 9	Describes the tasks required to create a custom test script type.
<i>Component Description and Communication Overview</i> on page 10	Lists test script adapter components and presents a diagram describing their interaction during playback.
<i>Using the Command Line Execution Engine</i> on page 12	Explains how to use <code>rttsee</code> to execute test scripts from a command line.

Built-In Test Script Types

When a TestManager user plays back (executes) a test script, test case, or suite, the playback component of TestManager, the *Test Script Execution Engine* (TSEE), calls the *Test Script Execution Adapter* (TSEA) associated with each type of test script in the suite. As released, TestManager supports execution of the *test script types* listed and described in the table on the next page.

Test Script Type	Description
GUI	A functional test script written in SQA Basic, a proprietary Basic-like scripting language.
VU	A performance test script written in VU, a proprietary C-like scripting language.
VB	A test script written in the Visual Basic language — for example, a test script for an application based on Microsoft's Component Object Model (COM).
Java	A test script written in the Java language — for example, a test script for a BEA WebLogic application.
Command Line	A test script (written in any language) that can be executed from the command line — for example, a DOS batch file, a Perl or Bourne shell script, or compiled C program.
Manual	A procedure explaining how to perform a test manually that, when executed, prompts a tester to verify the result of the test.

Adding Test Script Types

There are two ways to extend TestManager to support a new test script type. One way is to create a test script type that is based on the Command Line test script type and that uses the Command Line Test Script Console Adapter (TSCA) provided with TestManager. The procedure for doing this is explained in *Adding a Command Line Test Script Type* on page 3. The advantage of this method is simplicity: it requires no custom programming. The only requirement is that the test scripts you want to run from TestManager can be executed from the command line. The drawback of these scripts is that, while TestManager can execute them individually and also in suites that include test scripts of other types, the scripts are not fully integrated into TestManager. For example:

- Test scripts that use the Command Line adapter require separate process invocations each time they are run. Custom test scripts do not, and so run more efficiently.
- Any procedures (such as compilation/linking) required to make these test scripts executable must be performed outside of TestManager.

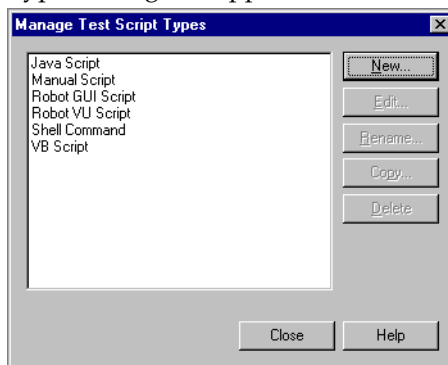
If you extend TestManager in this way, see *The Command Line Interface to Rational Test Script Services*. This manual explains `tsscmd`, a utility that gives Command Line test scripts access to Rational Test Script Services. (The C bindings for these services are documented in Chapter 3). Command-line test scripts that do not use Test Script Services can be executed from TestManager but are not integrated into the TestManager logging, monitoring, and reporting framework.

The other way to extend TestManager is to create a custom test script type. This method, described in *Adding a Custom Test Script Type* on page 9, requires that you develop programs implementing the C APIs described in chapters 2 and 4 of this manual and, optionally, provide access to the Test Script Services documented in Chapter 3. For example, you can create adapters for currently unsupported scripting languages or software testing environments. Custom test script types are fully integrated into the TestManager framework, but they require considerably more effort to provide.

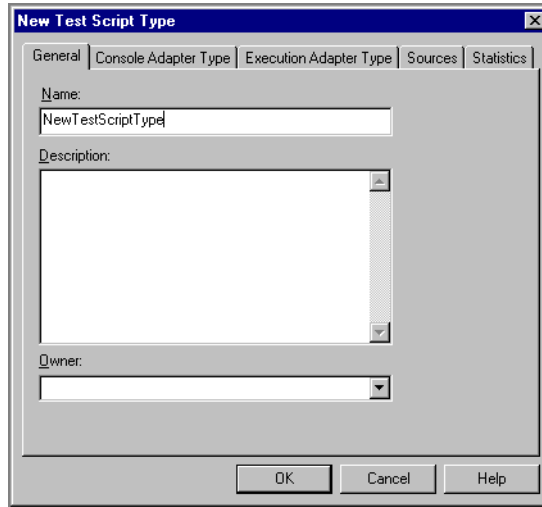
Adding a Command Line Test Script Type

This example illustrates how, without doing any programming, to extend TestManager so that it can support test scripts written in a new source language. After you have followed these procedures, TestManager can manage test scripts written in Perl. Specifically, a TestManager user can view, edit, or play back Perl source test scripts. Additionally, Perl test scripts can be added to TestManager suites that include test scripts of other types.

- 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.

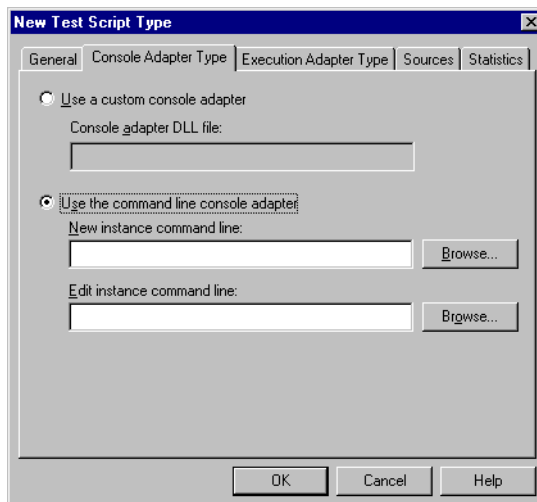


- 3 Click the **New** button. The New Test Script Type dialog box appears with the **General** tab selected.



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:

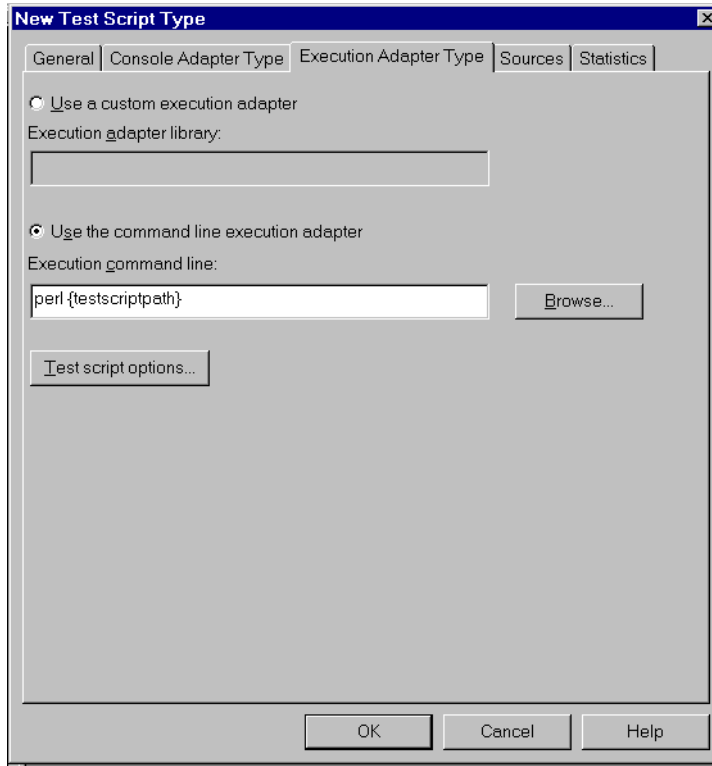
```
notepad
```
- In the **Edit instance command line** box, type the command to start in order to view or edit existing scripts of this type. For example:

```
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.

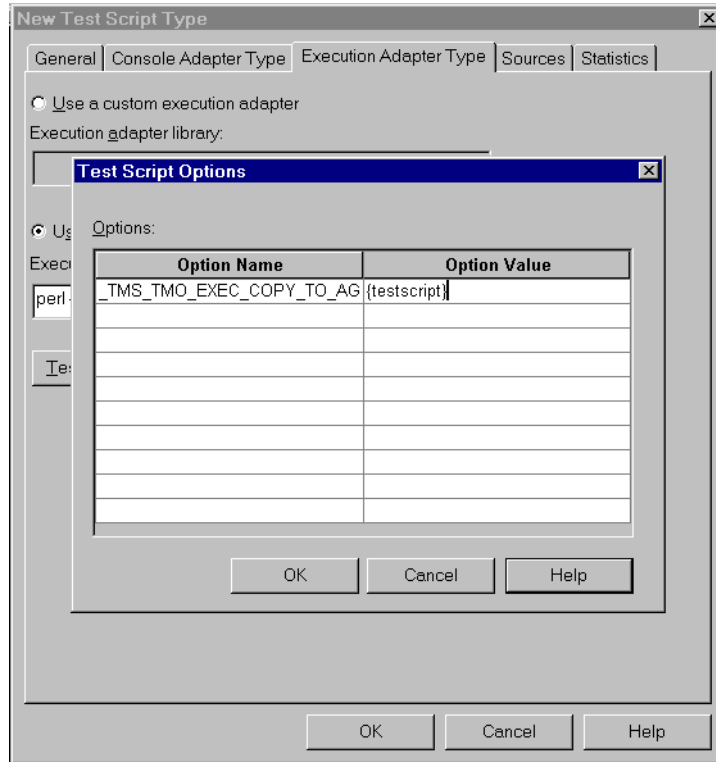


Click **Use the command line execution adapter**. In the **Execution command line** box, type the execution command line for a new script instance. In this example, type the following exactly as shown:

```
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 the **Test Script Options** button. The Test Script Options dialog box opens as shown below.



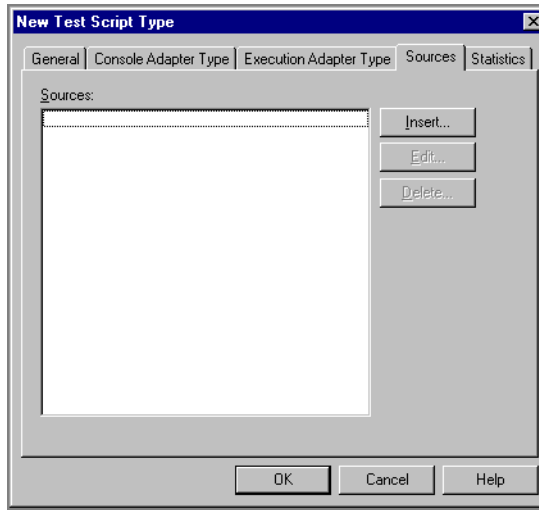
In the **Options** area, type the following Option Name and Option Value pair:

Option Name: `_TMS_TSO_EXEC_COPY_TO_AGENT_FILELIST`

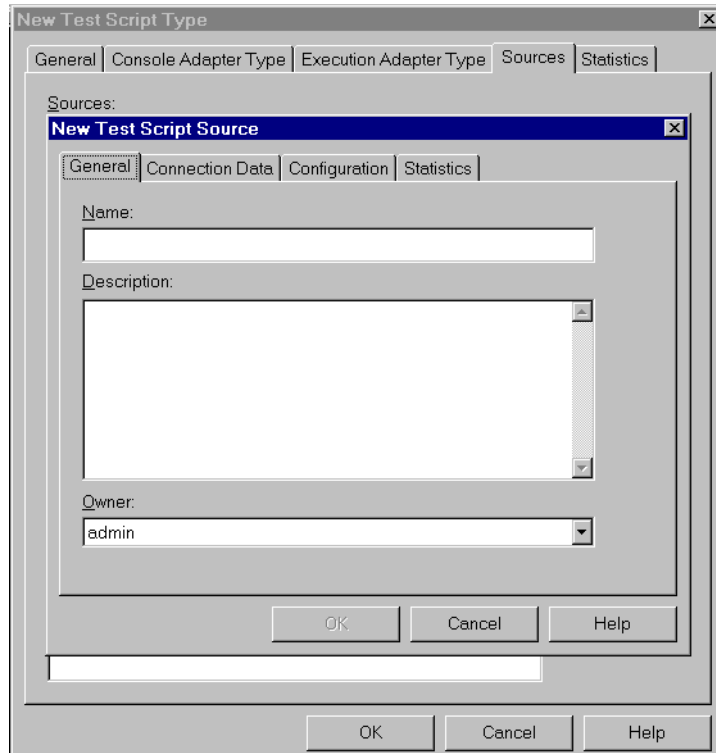
Option Value: `{testscript}`

Click OK.

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



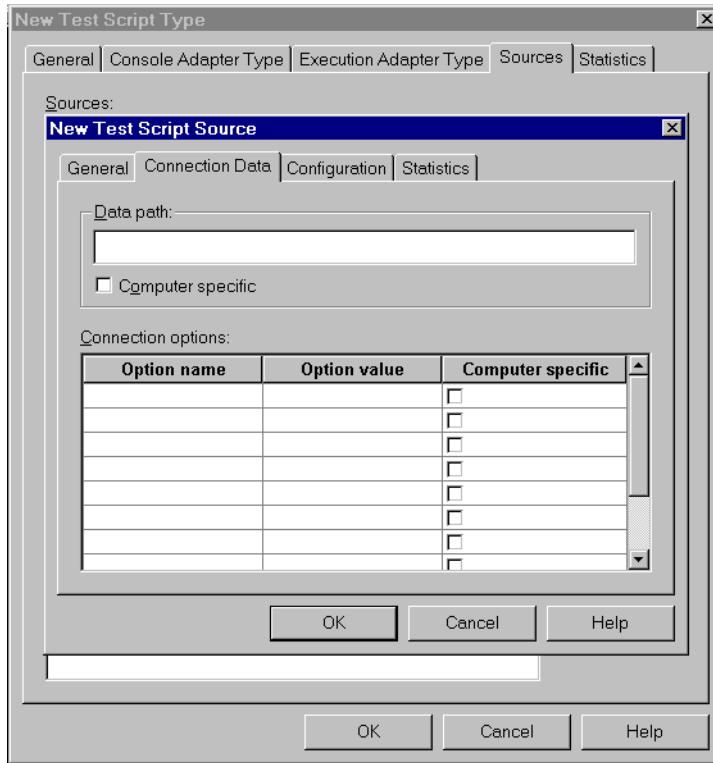
- Click the **Insert** button. A popup appears telling you that the test script you are defining must be created before proceeding — answer **Yes**. The dialog box changes as shown below.



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 will be added to TestManager's **File > New Test Script**, **File > Open Test Script**, and **File > Run Test Script** drop-down lists. You will select this name to create a new Perl script or edit/view/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 will be 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.

Adding a Custom Test Script Type

The tasks required to support a new custom test script type are as follows:

- Write a program that supports the Test Script Execution Adapter (TSEA) API described in Chapter 2 of this manual.

The TSEA API is the interface that allows TestManager to call a TSEA for a particular test script type.

- Add support for the *test script services* (TSS) described in Chapter 3.

The TSS API gives scripts of the new type access to services such as the following:

- Use of datapools to provide meaningful test data to scripts during execution
 - Insertion of timers and synchronization points
 - Monitoring of script playback progress
 - Logging for analysis and reporting
 - Exchange of information among virtual testers through environment, internal, and shared variables
- Write a Test Script Console Adapter (TSCA) for the new test script type as described in Chapter 4.

The TSCA allows TestManager to locate scripts of the new type and associated programs needed to manipulate the scripts.

- Register adapter components with TestManager.

Create the new test script type and give TestManager the names and locations of the TSCA, the TSEA, and the programs to be used to edit or view scripts of the new type. These procedures are explained in the TestManager online Help and in the *Rational TestManager User's Guide*. The procedures are similar to those described in *Adding a Command Line Test Script Type* on page 3.

Your code must reside in dynamic-link libraries (.dll in Windows or NT, or .so in UNIX). During initialization, the TestManager TSEE dynamically links with the TSEA component of your adapter.

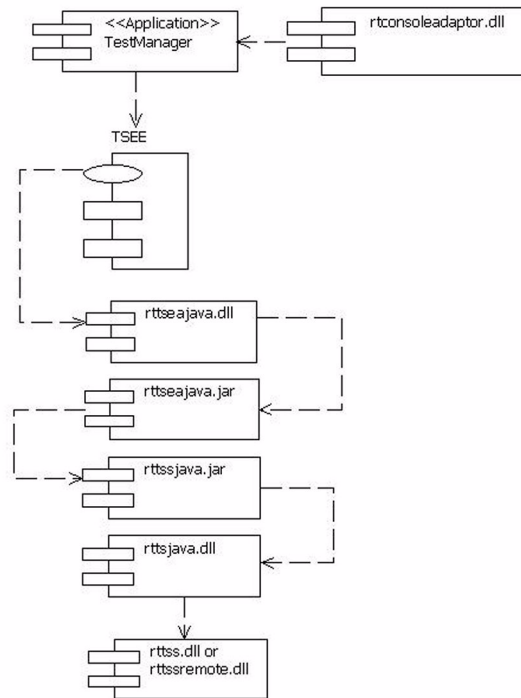
TSEA dlls must be placed in the Rational Test\tsea folder under the Rational installation directory.

Component Description and Communication Overview

The following table lists and describes the test script adapter components on Windows NT systems. (UNIX systems have components of the same names but with appropriate suffixes.)

File	Path in Installation Directory	Description
rtss.dll	Rational Test\	The C Test Script Services library. Used for scripts executed inside TSEE process space.
rtssremote.dll	Rational Test\	The C proxy Test Script Services library. Used for scripts executed by a proxy server started by the TSEA.
rttsee.exe	Rational Test\	The command line TSEE. Used for testing a TSEA or TSS implementation.
tsea	Rational Test\	Directory where TSEA components must reside.
rtss.lib	Rational Test\rtsdk\c\lib\	The TSS LIB file for linking.
rtssremote.lib	Rational Test\rtsdk\c\lib\	The TSS proxy LIB file for linking.
rtss.h	Rational Test\rtsdk\c\include\	The header file defining the TSS calling interface.
tsea.h	Rational Test\rtsdk\c\include\	The header file defining the TSEA calling interface.
testtypeapi.h	Rational Test\rtsdk\c\include\	The header file defining the TSCA calling interface.

The following Rational Rose® diagram illustrates how the test script adapter components described in the previous table work together, using the Rational Java adapter components as an example.



A TestManager suite can contain many test scripts of different types. When a user runs a suite, TestManager relies on the Test Script Console Adapter (TSCA, described in Chapter 4) to locate scripts of the supported types, and to associate each script type with the program(s) used to edit or view the scripts. The TSCA used for Java test script types is `rtconsoleadaptor.dll`, located in the Rational Test folder under the Rational installation directory.

To play back test scripts, TestManager starts a Test Script Execution Adapter (TSEA) that knows how to execute each test script type. The Java TSEA is `rttseajava.dll`, located in the Rational Test\tsea folder under the Rational installation directory.

The design of a TSEA differs depending on the languages involved and on its scope. The Java TSEA in the diagram has four components:

- A C component (`rttseajava.dll`) implementing the TSEA API (described in Chapter 2). This is the component that receives and responds to calls from the TSEE and initializes a Java virtual machine. A session is opened and one or more tasks (scripts of type Java) are created. The TSEE remains in contact with the TSEA C component until the session is complete.
- A Java component (`rttseajava.jar`) that executes Java scripts, which may include calls to the `rttssjava.jar` component.
- A Java class library (`rttssjava.jar`) implementing, in Java, the TSS C library functions in `rttss.dll` (Chapter 3).
- A C component (`rttssjava.dll`). Calls from `rttssjava.jar` to `rttss.dll` go through this layer, which converts between Java and C data types and structures.

At the bottom of the diagram is the Test Script Services library, implemented by a dynamic-link C library. This is the layer where requested services are performed and integrated into the TestManager UI. The Java TSEA can be linked with `rttss.dll` for direct script execution or with `rttssremote.dll` for proxy execution.

Using the Command Line Execution Engine

The command-line execution engine, `rttsee`, lets you test your TSEA from the command line rather than from TestManager. The `rttsee` interface is especially useful on non-Windows platforms, and for testing your extension of the TSEE framework independently of the test scripts executed through the framework.

The following example illustrates the most common usage of `rttsee`. It runs a Java program named `hello.java` via the Java TSEA, `rttseajava.dll`.

```
rttsee -e rttseajava hello
```

The following example starts a TSS server listening on port 95 that continues running until explicitly stopped.

```
rttsee -k -P 95
```

The syntax of `rttsee` is:

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

The full options are described in the following table.

Option	Description
<code>-d dir</code>	Specifies the directory for result files — u-file (log), o-file, e-file. The default is the current directory.
<code>-e tsea[:type]</code> <code>script[:type]</code>	Specifies the TSEA to start and the test script to run. If <i>tsea</i> handles test scripts of more than one type, <i>type</i> indicates the type of <i>script</i> . The <i>type</i> may be specified with either or both the TSEA or script, but it must match if specified with both.
<code>-G [I i T t]</code>	Controls random number generation. Enter one choice (I or i, T or t) from either or both pairs: <ul style="list-style-type: none"> ▪ I Generate unique seeds for each virtual tester, using either the predefined seed or one specified with <code>-S</code> (default). ▪ i Use the same seed for all virtual testers, either the predefined seed or one specified with <code>-S</code>. ▪ t Seed the generator once for all tasks at the beginning, using either the predefined seed or one specified with <code>-S</code> (default). ▪ T Reseed the generator at the beginning of each task.
<code>-k</code>	Keep-alive. Use with <code>-P</code> to start a TSS server that keeps running after all test scripts have completed execution.
<code>-P portnumber</code>	Specifies the listening port for a TSS server that remains alive until explicitly stopped.
<code>-r</code>	Redirects stdio to the o-file and e-file (in the directory specified by <code>-d</code>).
<code>-S seed</code>	Specifies an alternative seed value for the predefined seed. Must be a positive integer except in conjunction with <code>-G i</code> .
<code>-u uid</code>	Specifies the ID of a virtual tester.
<code>-V</code>	Displays the <code>rttsee</code> version.

About This API

This chapter describes the Rational Test Script Execution Adapter (TSEA) API. This API defines the C language calls that the TestManager Test Script Execution Engine (TSEE) uses to communicate with a Test Script Execution Adapter (TSEA). Your TSEA must respond to these calls as described in this chapter.

Communication Overview

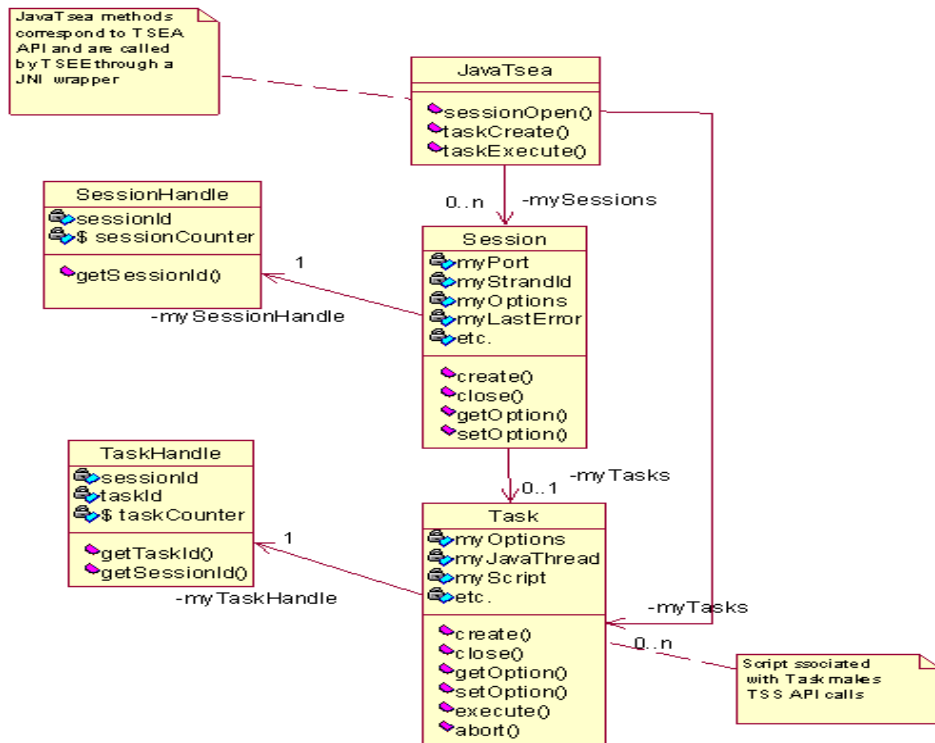
Communication between the TSEE and the TSEA occurs in three phases as described in the following steps.

- 1** Initialization phase. The TSEE:
 - a** Dynamically links in the DLL for your TSEA (located under the installation folder in Rational Test\tsea).
 - b** Calls `SessionOpen()` to start a session; the TSEA returns a session handle.
 - c** Optionally, calls `SessionSetOption()` to set one or more session options. Session options apply to all tasks (scripts) in a session. An option may be anything (such as a working directory or timer) needed during execution.
- 2** Execution phase. The TSEE:
 - a** Calls `TaskCreate()`, which creates a test script of a type that the TSEA knows how to execute; the TSEA returns a task handle.
 - b** Optionally, calls `TaskSetOption()` to set one or more task options.
 - c** Calls `TaskExecute()`; the TSEA executes the task and upon completion returns the status.
 - d** Closes the task handle.
 - e** Repeats a–d until all tasks for this TSEA have been completed.

3 Cleanup phase. The TSEE calls `SessionClose()` to close the session.

At any time during task execution, the TSEE might call `TaskAbort()`. For example, if the TestManager user choses to stop an executing test script or suite run, the TSEE calls `TaskAbort()`. If this happens, `TaskExecute()` should return as soon as possible with a termination status. The TSEE will then terminate the session as cleanly as possible.

The following diagram (generated using Rose) is a static diagram illustrating the interactions among the components of the Java TSEA provided with TestManager.



Data Types and C Header Files

The following table lists and describes the TSEA data types. Defined in the header file `rtss.h`, these are the types of data that your TSEA receives from and returns to the TSEE.

Type	Description
<code>s32</code>	Signed 32-bit integer.
<code>u16</code>	Unsigned 16-bit integer.
<code>SessionHandle</code>	Returned to the TSEE after a successful <code>SessionOpen()</code> call, and included with session calls and <code>TaskCreate()</code> .
<code>TaskHandle</code>	Returned to the TSEE after a successful <code>TaskCreate()</code> call, and included with all other task calls.
<code>TaskType</code>	Returned to the TSEE with a successful <code>TaskCreate()</code> call, specifying the TSEA's test script type.

Summary

The TSEA API includes the following calls.

Function	Description
<code>SessionClose()</code>	Closes a TSEA session.
<code>SessionGetOption()</code>	Gets TSEA session options.
<code>SessionOpen()</code>	Opens a TSEA session.
<code>SessionSetOption()</code>	Sets TSEA session options.
<code>TaskAbort()</code>	Aborts a task.
<code>TaskClose()</code>	Closes a TSEA task.
<code>TaskCreate()</code>	Opens a TSEA task.
<code>TaskExecute()</code>	Executes a task.
<code>TaskGetOption()</code>	Gets TSEA task options.
<code>TaskSetOption()</code>	Sets TSEA task options.
<code>TSEAEError()</code>	Gets TSEA error information.

SessionClose()

SessionClose()

Closes a TSEA session.

Syntax

```
s32 SessionClose (SessionHandle session)
```

Element	Description
<i>session</i>	The handle of the TSEA session to close.

Comments

TSEE makes this call when the last script of a playback request has completed. Your TSEA should perform any cleanup necessitated by the run.

See Also

SessionOpen()

SessionGetOption()

Gets the value of a session option.

Syntax

```
s32 SessionGetOption (SessionHandle session, char *optname,  
void *optval, s32 len)
```

Element	Description
<i>session</i>	The session handle, returned by SessionOpen().
<i>optname</i>	The session option whose value is to be returned.
<i>optval</i>	The value of <i>optname</i> that is returned to the TSEE.
<i>len</i>	Storage buffer size. When TSEE makes the call, <i>optval</i> is an empty buffer of this size; on return, <i>len</i> is the size of the value pointed to by <i>optval</i> .

See Also

`SessionSetOption()`

SessionOpen()

Opens a session with a TSEA.

Syntax

```
SessionHandle SessionOpen (char *hostname, u16 port, s32
    strandID, char **message)
```

Element	Description
<i>hostname</i>	The name (or IP address in dot notation) of the TSEA host.
<i>port</i>	The listening port used by the TSEE for communication with proxy TSS processes. Not used by scripts that are directly executed by TSEE. Where a TSEA uses <code>TSSConnect()</code> to start a proxy script execution process, this <i>port</i> must be passed to the process.
<i>strandID</i>	Strand (thread) ID for TSS calls in this session.
<i>message</i>	A statement that, if the open fails, is included with the log.

Comments

On success, return to the TSEE a unique session identifier of type `SessionHandle`.
On failure, return NULL. If NULL is returned, the failure is logged.

See Also

`SessionClose()`

SessionSetOption()

Sets the value of a session option.

Syntax

```
s32 SessionSetOption (SessionHandle session, char *optname,
    void *optval, s32 len)
```

Element	Description
<i>session</i>	The session handle, returned by <i>SessionOpen()</i> .
<i>optname</i>	The session option whose value is to be set.
<i>optval</i>	The new value of <i>optname</i> .
<i>len</i>	The size of buffer <i>optval</i> .

See Also

SessionGetOption()

TaskAbort()

Aborts a TSEA task.

Syntax

```
s32 TaskAbort (TaskHandle task)
```

Element	Description
<i>task</i>	The handle of the TSEA task to abort.

Comments

The TSEE makes this call (from another thread) to abort a task. Your TSEA should stop the task run as soon as possible and return a value greater than 0 indicating that the task has been aborted.

See Also

`TaskClose()`, `TaskCreate()`, `TaskExecute()`

TaskClose()

Closes a TSEA task.

Syntax

```
s32 TaskClose (TaskHandle task)
```

Element	Description
<i>task</i>	The handle of the TSEA task to close.

Comments

The TSEE makes this call when a task completes. Your TSEA should perform any cleanup necessitated by the task execution.

See Also

`TaskAbort()`, `TaskCreate()`, `TaskExecute()`

TaskCreate()

Creates a task.

Syntax

```
TaskHandle TaskCreate (SessionHandle session, TaskType type,  
char *sourcelocation, char *testScriptId)
```

Element	Description
<i>session</i>	The session handle, returned by <code>SessionOpen()</code> .
<i>type</i>	The test script type for scripts that this TSEA plays back.
<i>sourcelocation</i>	The location where source scripts of <i>type</i> are located.
<i>testScriptId</i>	The name of the file containing the test script.

TaskExecute()

Comments

On success, return to the TSEE a unique task identifier of type TaskHandle. On failure, return NULL.

See Also

TaskAbort(), TaskClose(), TaskExecute()

TaskExecute()

Executes a TSEA task.

Syntax

```
s32 TaskExecute (TaskHandle task)
```

Element	Description
<i>task</i>	The handle of the TSEA task to execute.

Comments

The TSEE makes this call to execute a task. Your TSEA should return 0 if the task completes successfully or a number greater than 0 if the task fails.

See Also

TaskAbort(), TaskClose(), TaskCreate()

TaskGetOption()

Gets the value of a task option.

Syntax

```
s32 TaskGetOption (TaskHandle task, char *optname, void  
*optval, s32 len)
```

Element	Description
<i>task</i>	The task handle, returned by <code>TaskCreate()</code> .
<i>optname</i>	The task option whose value is to be returned.
<i>optval</i>	The value of <i>optname</i> that is returned to the TSEE.
<i>len</i>	Storage buffer size. When TSEE makes the call, <i>optval</i> is an empty buffer of this size; on return, <i>len</i> is the size of the value pointed to by <i>optval</i> .

See Also

`TaskSetOption()`

TaskSetOption()

Sets the value of a task option.

Syntax

```
s32 TaskSetOption (TaskHandle task, char *optname, void
    *optval, s32 len)
```

Element	Description
<i>task</i>	The task handle, returned by <code>SessionOpen()</code> .
<i>optname</i>	The task option whose value is to be set.
<i>optval</i>	The new value of <i>optname</i> .
<i>len</i>	The size of buffer <i>optval</i> .

See Also

`TaskGetOption()`

TSEError()

TSEError()

Gets a message following an error.

Syntax

```
s32 TSEError (SessionHandle session, char **message)
```

Element	Description
<i>session</i>	The session handle, returned by <i>SessionOpen()</i> .
<i>message</i>	String explaining the cause of a TSEA call failure.

Comments

The TSEE makes this call whenever a TSEA call returns a value greater than 0. Your TSEA should allocate a message buffer for each open session and supply a message indicating the cause of a failure.

About Test Script Services

This chapter describes the Rational Test Script Services (TSS). These are the services that can be extended to languages other than C. If you wrap these calls in the language provided by your Test Script Execution Adapter, these services are available to test script developers in that language. The services 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 Services

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 functions listed in the following table to access and manipulate datapools within your scripts.

Function	Description
<code>TSSDatapoolClose ()</code>	Closes a datapool.
<code>TSSDatapoolColumnCount ()</code>	Returns the number of columns in a datapool.
<code>TSSDatapoolColumnName ()</code>	Returns the name of the specified datapool column.
<code>TSSDatapoolFetch ()</code>	Moves the datapool cursor to the next row.
<code>TSSDatapoolOpen ()</code>	Opens the named datapool and sets the row access order.
<code>TSSDatapoolRewind ()</code>	Resets the datapool cursor to the beginning of the datapool access order.
<code>TSSDatapoolRowCount ()</code>	Returns the number of rows in a datapool.
<code>TSSDatapoolSearch ()</code>	Searches a datapool for the named column with a specified value.
<code>TSSDatapoolSeek ()</code>	Moves the datapool cursor forward.
<code>TSSDatapoolValue ()</code>	Retrieves the value of the specified datapool column.

TSSDatapoolClose()

Closes a datapool.

Syntax

```
s32 TSSDatapoolClose (s32 dpid)
```

Element	Description
<i>dpid</i>	The ID of the datapool to close. Returned by TSSDatapoolOpen().

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_INVALID. The datapool identifier is invalid.

Example

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

```
s32 dpid = TSSDatapoolOpen ("custdata", 0, 0, NULL);
if (dpid > 0)
    s32 retVal = TSSDatapoolClose (dpid);
```

See Also

TSSDatapoolOpen()

TSSDatapoolColumnCount()

Returns the number of columns in a datapool.

Syntax

```
s32 TSSDatapoolColumnCount (s32 dpid)
```

Element	Description
<i>dpid</i>	The ID of the datapool. Returned by TSSDatapoolOpen().

Return Value

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

- TSS_OK. Success.
- TSS_NOSERVER. No previous successful call to TSSConnect ().
- TSS_INVALID. The datapool identifier is invalid.
- TSS_ABORT. 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.

```
s32 dpid = TSSDatapoolOpen ("custdata", 0, 0, NULL);
if (dpid > 0)
    s32 columns = TSSDatapoolColumnCount (dpid);
```

TSSDatapoolColumnName()

Gets the name of the specified datapool column.

Syntax

```
char * TSSDatapoolColumnName (s32 dpid, s32 columnNumber)
```

Element	Description
<i>dpid</i>	The ID of the datapool. Returned by TSSDatapoolOpen ().
<i>columnNumber</i>	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 function returns the name of the specified datapool column. The function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSERVER. No previous successful call to TSSConnect ().
- TSS_INVALID. The datapool identifier or column number is invalid.
- TSS_ABORT. 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.

```
char *colName;
s32 dpid = TSSDatapoolOpen("custdata",0,0,NULL);
if (dpid > 0)
    if (TSSDatapoolFetch(dpid) == TSS_OK)
        colName = TSSDatapoolColumnName(dpid,3);
```

TSSDatapoolFetch()

Moves the datapool cursor to the next row.

Syntax

```
s32 TSSDatapoolFetch (s32 dpid)
```

Element	Description
<i>dpid</i>	The ID of the datapool. Returned by TSSDatapoolOpen().

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_EOF. The end of the datapool was reached.
- TSS_NOSERVER. No previous successful call to TSSConnect().
- TSS_INVALID. The datapool identifier is invalid.
- TSS_ABORT. 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 TSSDatapoolValue().

The “next row” is determined by the *assessFlags* passed with the open call. The default is the next row in sequence. See TSSDatapoolOpen().

After a datapool is opened, a TSSDatapoolFetch() is required before the initial row can be accessed.

TSSDatapoolOpen()

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

Example

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

```
s32 dpid = TSSDatapoolOpen ("custdata", 0, 0, NULL);  
if (dpid > 0)  
    s32 retVal = TSSDatapoolFetch (dpid);
```

See Also

TSSDatapoolOpen(), TSSDatapoolSeek(), TSSDatapoolValue()

TSSDatapoolOpen()

Opens the named datapool and sets the row access order.

Syntax

```
s32 TSSDatapoolOpen(char *name, u32 accessFlags, s32  
    overrideCount, NamedValue *overrides)
```

Element	Description
<i>name</i>	The name of the datapool to open. If <i>accessFlags</i> includes TSS_DP_NO_OPEN, no CSV datapool is opened; instead, <i>name</i> refers to the contents of <i>overrides</i> specifying a one-row table. Otherwise, the CSV file <i>name</i> in the Rational project is opened.

Element	Description
<i>accessFlags</i>	<p>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:</p> <ol style="list-style-type: none"> 1 Specify the sequence in which datapool rows are accessed: <ul style="list-style-type: none"> TSS_DP_SEQUENTIAL – physical order (default) TSS_DP_RANDOM – any order, including multiple access or no access TSS_DP_SHUFFLE – access order is shuffled after each access 2 Specify what happens after the last datapool row is accessed: <ul style="list-style-type: none"> TSS_DP_NOWRAP – end access to the datapool (default) TSS_DP_WRAP – go back to the beginning 3 Specify whether the datapool cursor is shared by all virtual testers or is unique to each: <ul style="list-style-type: none"> TSS_DP_PRIVATE – virtual testers each work from their own sequential, random, or shuffle access order (default) TSS_DP_SHARED – all virtual testers work from the same access order 4 TSS_DP_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 TSS_DP_PRIVATE. Ignored if used with TSS_DP_RANDOM. 5 TSS_DP_REWIND specifies that the datapool should be rewound when opened. Ignored unless used with TSS_DP_PRIVATE. 6 TSS_DP_NO_OPEN specifies that, instead of a CSV file, the opened datapool consists only of column/value pairs specified in a local array <i>overrides</i>[].
<i>overrideCount</i>	The number of columns in array <i>overrides</i> . Must be greater than 0 if access flag TSS_DP_NO_OPEN is specified; otherwise, must be 0.
<i>overrides</i>	A local, two-dimensional array of column/value pairs, where <i>overrides</i> [n].name is the column name and <i>overrides</i> [n].value is the value returned by TSSDatapoolValue() for that column name. Unless access flag TSS_DP_NO_OPEN is present, specify as NULL.

Return Value

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

- `TSS_OK`. Success.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_INVALID`. The *accessFlags* argument is or result in an invalid combination.
- `TSS_NOTFOUND`. No datapool of the given *name* was found.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

Comments

If the *accessFlags* argument is specified as 0, 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 `TSS_DP_SHARED` option initializes the cursor. In the same suite run (and, with the `TSS_DP_PERSIST` flag, in subsequent suite runs), virtual testers that subsequently use the same datapool opened with `TSS_DP_SHARED` share the initialized cursor.
- The first open that uses the `TSS_DP_PRIVATE` option initializes the private cursor for a virtual tester. In the same suite run, a subsequent open that uses `TSS_DP_PRIVATE` sets the cursor to the last row accessed by that virtual tester.

The `NamedValue` data type is defined as follows:

```
typedef struct {
    char *Name;
    char *Value;
} NamedValue;
```

Example

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

```
s32 dpid = TSSDatapoolOpen ("custdata",TSS_DP_SHUFFLE |
TSS_DP_PERSIST,0,NULL);
```

See Also

`TSSDatapoolClose()`

TSSDatapoolRewind()

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

Syntax

```
s32 TSSDatapoolRewind (s32 dpid)
```

Element	Description
<i>dpid</i>	The ID of the datapool. Returned by <code>TSSDatapoolOpen()</code> .

Return Value

This function exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_INVALID`. The datapool identifier is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

Comments

The datapool is rewound as follows:

- For datapools opened `DP_SEQUENTIAL`, `TSSDatapoolRewind()` resets the cursor to the first record in the datapool file.
- For datapools opened `DP_RANDOM` or `DP_SHUFFLE`, `TSSDatapoolRewind()` restarts the random number sequence.
- For datapools opened `DP_SHARED`, `TSSDatapoolRewind()` has no effect.

TSSDatapoolRowCount()

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.

```
s32 dpid = TSSDatapoolOpen ("custdata", 0, 0, NULL);
if (dpid > 0)
{
    TSSDatapoolSeek (dpid, 2);
    TSSDatapoolRewind (dpid);
};
```

TSSDatapoolRowCount()

Returns the number of rows in a datapool.

Syntax

```
s32 TSSDatapoolRowCount (s32 dpid)
```

Element	Description
<i>dpid</i>	The ID of the datapool. Returned by <code>TSSDatapoolOpen()</code> .

Return Value

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

- `TSS_OK`. Success.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_INVALID`. The datapool identifier is invalid.
- `TSS_ABORT`. 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.

```
s32 dpid = TSSDatapoolOpen ("custdata", 0, 0, NULL);
if (dpid > 0)
    s32 rows = TSSDatapoolRowCount (dpid);
```

TSSDatapoolSearch()

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

Syntax

```
s32 TSSDatapoolSearch(s32 dpid, s32 keyCount, NamedValue *keys)
```

Element	Description
<i>dpid</i>	The ID of the datapool. Returned by <code>TSSDatapoolOpen()</code> .
<i>keycount</i>	The number of columns in <i>keys</i> .
<i>keys</i>	An array containing values to be searched for.

Return Value

This function exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_EOF`. The end of the datapool was reached.
- `TSS_NOSERVER`. No previous successful call to `TSSConnect()`.
- `TSS_INVALID`. The datapool identifier is invalid.
- `TSS_ABORT`. 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.

The `NamedValue` data type is defined as follows:

```
typedef struct {
    char *Name;
    char *Value;
} NamedValue;
```

Example

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

```
NamedValue toFind[1];
toFind[0].Name = "Last";
toFind[0].Value = "Doe";
s32 dpid = TSSDatapoolOpen("custdata",0,0,NULL);
if (dpid > 0)
    if (TSSDatapoolFetch(dpid) == TSS_OK)
        s32 rowNumber = TSSDatapoolSearch(dpid,1,toFind);
```

TSSDatapoolSeek()

Moves the datapool cursor forward.

Syntax

```
s32 TSSDatapoolSeek (s32 dpid, s32 count)
```

Element	Description
<i>dpid</i>	The ID of the datapool. Returned by <code>TSSDatapoolOpen()</code> .
<i>count</i>	A positive number indicating the number of rows to move forward in the datapool.

Return Value

This function exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_EOF`. The end of the datapool was reached.
- `TSS_NOSERVER`. No previous successful call to `TSSConnect()`.
- `TSS_INVALID`. The datapool identifier is invalid.
- `TSS_ABORT`. 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 `TSSDatapoolValue()`.

The meaning of “forward” depends on the *accessFlags* passed with the open call; see `TSSDatapoolOpen()`. This call is functionally equivalent to calling `TSSDatapoolFetch()` *count* times.

An end-of-file (TSS_EOF) error results if cursor wrapping is disabled (by access flag TSS_DP_NOWRAP) and *count* moves the access row beyond the last row. If `TSSDatapoolValue()` 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.

```
s32 dpid = TSSDatapoolOpen ("custdata", 0, 0, NULL);
if (dpid > 0)
    TSSDatapoolSeek (dpid, 2);
```

See Also

`TSSDatapoolFetch()`, `TSSDatapoolOpen()`, `TSSDatapoolValue()`

TSSDatapoolValue()

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

Syntax

```
char * TSSDatapoolValue(s32 dpid, char *columnName)
```

Element	Description
<i>dpid</i>	The ID of the datapool. Returned by <code>TSSDatapoolOpen()</code> .
<i>columnName</i>	The name of the column whose value you want to retrieve.

Return Value

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

- TSS_OK. Success.
- TSS_EOF. The end of the datapool was reached.
- TSS_NOSEVER. No previous successful call to `TSSConnect()`.

- `TSS_INVALID`. The specified *columnName* is not a valid column in the datapool.
- `TSS_ABORT`. 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 will have been loaded into memory either by `TSSDatapoolFetch()` or `TSSDatapoolSeek()`.

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 `TSS_DP_NO_OPEN` access flag, the returned value comes from an override list provided with the open call.

Example

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

```
char *colVal;
s32 dpid = TSSDatapoolOpen("custdata", 0, 0, NULL)
if (dpid > 0)
    if (TSSDatapoolFetch(dpid) == TSS_OK)
        colVal = TSSDatapoolValue("Middle");
```

See Also

`TSSDatapoolFetch()`, `TSSDatapoolOpen()`, `TSSDatapoolSeek()`

Logging Services

Use the logging functions 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 `EVAR_LogEvent_control` 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 call). Use the environment variable `EVAR_LogData_control` to control whether or not any data associated with an event is logged.

Summary

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

Function	Description
TSSLogEvent ()	Logs an event.
TSSLogMessage ()	Logs a message event.
TSSLogTestCaseResult ()	Logs a test case event.

TSSLogEvent()

Logs an event.

Syntax

```
s32 TSSLogEvent (char *eventType, s16 result, char
    *description, s32 propertyCount, NamedValue *property)
```

Element	Description
<i>eventType</i>	Contains the description to be displayed in the log for this event.
<i>result</i>	Specifies the notification preference regarding the result of the call. Can be one of the following: <ul style="list-style-type: none"> ▪ TSS_LOG_RESULT_NONE (default: no notification) ▪ TSS_LOG_RESULT_PASS ▪ TSS_LOG_RESULT_FAIL ▪ TSS_LOG_RESULT_WARN ▪ TSS_LOG_RESULT_STOPPED ▪ TSS_LOG_RESULT_INFO ▪ TSS_LOG_RESULT_COMPLETED ▪ TSS_LOG_RESULT_UNEVALUATED 0 specifies the default.
<i>description</i>	Contains the string to be put in the entry's failure description field.
<i>propertyCount</i>	Specifies the number of rows in the property array.

TSSLogEvent()

Element	Description
<i>property</i>	An array containing property name/value pairs, where <code>property [n] . name</code> is the property name and <code>property [n] . value</code> is its value.

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSEVER. No previous successful call to TSSConnect ().
- TSS_INVALID. An unknown *result* was specified.
- TSS_ABORT. 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 `EVAR_LogData_control` or `EVAR_LogEvent_control` environment variables. Alternatively, the logging preference can be set with the `EVAR_Log_level` and `EVAR_Record_level` environment variables. The `TSS_LOG_RESULT_STOPPED`, `TSS_LOG_RESULT_COMPLETED`, and `TSS_LOG_RESULT_UNEVALUATED` preferences are intended for internal use.

NamedValue is defined as follows:

```
typedef struct {
    char *Name;
    char *Value;
} NamedValue;
```

Example

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

```
NamedValue scriptProp[2];
scriptProp[0].Name = "ScriptName";
scriptProp[0].Value = "Login";
scriptProp[1].Name = "LineNumber";
scriptProp[1].Value = "1";
s32 retVal = TSSLogEvent("Login Dialog",0,"Login script failed",
2,scriptProp);
```

TSSLogMessage()

Logs a message.

Syntax

```
s32 TSSLogMessage(char *message, s16 result, char *description)
```

Element	Description
<i>message</i>	Specifies the string to log.
<i>result</i>	<p>Specifies the notification preference regarding the result of the call. Can be one of the following:</p> <ul style="list-style-type: none"> ▪ TSS_LOG_RESULT_NONE (default: no notification) ▪ TSS_LOG_RESULT_PASS ▪ TSS_LOG_RESULT_FAIL ▪ TSS_LOG_RESULT_WARN ▪ TSS_LOG_RESULT_STOPPED ▪ TSS_LOG_RESULT_INFO ▪ TSS_LOG_RESULT_COMPLETED ▪ TSS_LOG_RESULT_UNEVALUATED <p>0 specifies the default.</p>
<i>description</i>	Specifies the string to be put in the entry's failure description field.

Return Value

This function exits with one of the following results:

- TSS_OK.Success.
- TSS_NOSERVER.No previous successful call to TSSConnect () .
- TSS_ABORT.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 EVAR_LogData_control or EVAR_LogEvent_control environment variables.

TSSLogTestCaseResult()

Alternatively, the logging preference can be set with the `EVAR_Log_level` and `EVAR_Record_level` environment variables. The `TSS_LOG_RESULT_STOPPED`, `TSS_LOG_RESULT_COMPLETED`, and `TSS_LOG_RESULT_UNEVALUATED` preferences are intended for internal use.

Example

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

```
TSSLogMessage ("--Beginning of timed block T1--", 0, NULL);
```

TSSLogTestCaseResult()

Logs a test case result.

Syntax

```
s32 TSSLogTestCaseResult (char *testcase, s16 result, char  
    *description, s32 propertyCount, NamedValue *property[])
```

Element	Description
<i>testcase</i>	Identifies the test case whose result is to be logged.
<i>result</i>	Specifies the notification preference regarding the result of the call. Can be one of the following: <ul style="list-style-type: none">▪ <code>TSS_LOG_RESULT_NONE</code> (default: no notification)▪ <code>TSS_LOG_RESULT_PASS</code>▪ <code>TSS_LOG_RESULT_FAIL</code>▪ <code>TSS_LOG_RESULT_WARN</code>▪ <code>TSS_LOG_RESULT_STOPPED</code>▪ <code>TSS_LOG_RESULT_INFO</code>▪ <code>TSS_LOG_RESULT_COMPLETED</code>▪ <code>TSS_LOG_RESULT_UNEVALUATED</code> 0 specifies the default.
<i>description</i>	Contains the string to be displayed in the event of a log failure.
<i>propertyCount</i>	Specifies the number of rows in the property array.
<i>property</i>	An array containing property name/value pairs, where <code>property [n] . name</code> is the property name and <code>property [n] . value</code> is its value.

Return Value

This function exits with one of the following results:

- TSS_OK.Success.
- TSS_NOSEVER. No previous successful call to TSSConnect ().
- TSS_ABORT. 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 function 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 data associated settings in the EVAR_LogData_control or EVAR_LogEvent_control environment variables. Alternatively, the logging preference may be set by the EVAR_Log_level and EVAR_Record_level environment variables. The TSS_LOG_RESULT_STOPPED, TSS_LOG_RESULT_COMPLETED, and TSS_LOG_RESULT_UNEVALUATED preferences are intended for internal use.

The NamedValue data type is defined as follows:

```
typedef struct {
    char *Name;
    char *Value;
} NamedValue;
```

Example

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

```
NamedValue loginResult[1];
loginResult[0].Name = "Result";
loginResult[0].Value = "OK";
s32 retVal = TSSLogTestCaseResult("Verify login", 0, NULL,
1, loginResult);
```

Measurement Services

Use the measurement functions to set timers and environment variables and 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 functions.

Function	Description
TSSCommandEnd ()	Logs an end-command event.
TSSCommandStart ()	Logs a start-command event.
TSSEnvironmentOp ()	Sets an environment variable.
TSSGetTime ()	Gets the elapsed time of a run.
TSSInternalVarGet ()	Gets the value of an internal variable.
TSSThink ()	Sets a think-time delay.
TSSTimerStart ()	Marks the start of a block of actions to be timed.
TSSTimerStop ()	Marks the end of a block of timed actions.

TSSCommandEnd()

Marks the end of a timed command.

Syntax

```
s32 TSSCommandEnd(s16 result, char *description, s32 starttime,
  s32 endtime, char *logdata, s32 propertyCount, NamedValue
  *property)
```

Element	Description
<i>result</i>	Specifies the notification preference regarding the result of the call. Can be one of the following: <ul style="list-style-type: none"> ▪ TSS_LOG_RESULT_NONE (default: no notification) ▪ TSS_LOG_RESULT_PASS ▪ TSS_LOG_RESULT_FAIL ▪ TSS_LOG_RESULT_WARN ▪ TSS_LOG_RESULT_STOPPED ▪ TSS_LOG_RESULT_INFO ▪ TSS_LOG_RESULT_COMPLETED ▪ TSS_LOG_RESULT_UNEVALUATED. 0 specifies the default.
<i>description</i>	Contains the string to be displayed in the event of failure.
<i>starttime</i>	An integer indicating a time stamp to override the time stamp set by <code>TSSCommandStart()</code> . To use the time stamp set by <code>TSSCommandStart()</code> , specify as 0.
<i>endtime</i>	An integer indicating a time stamp to override the current time. To use the current time, specify as 0.
<i>logdata</i>	Text to be logged describing the ended command.
<i>propertyCount</i>	Specifies the number of rows in the property array.
<i>property</i>	An array containing property name/value pairs, where <code>property[n].name</code> is the property name and <code>property[n].value</code> is its value.

TSSCommandEnd()

Return Value

This function exits with one of the following results:

- TSS_OK.Success.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

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

An event and any data associated with it are logged only if the specified *result* preference matches associated settings in the EVAR_LogData_control or EVAR_LogEvent_control environment variables. Alternatively, the logging preference can be set with the EVAR_Log_level and EVAR_Record_level environment variables. The TSS_LOG_RESULT_STOPPED, TSS_LOG_RESULT_COMPLETED, and TSS_LOG_RESULT_UNEVALUATED preferences are intended for internal use.

The NamedValue data type is defined as follows:

```
typedef struct {
    char *Name;
    char *Value;
} NamedValue;
```

Example

This example marks the end of the timed activity specified by the previous TSSCommandStart() call.

```
s32 retVal = TSSCommandEnd(TSS_LOG_RESULT_PASS, "Command timer failed",
0, 0, "Login command completed", NULL);
```

See Also

TSSCommandStart(), TSSLogCommand()

TSSCommandStart()

Starts a timed command.

Syntax

```
s32 TSSCommandStart (char *label, char *name, RunState state)
```

Element	Description
<i>label</i>	The name of the timer to be started and logged, or NULL for an unlabeled timer.
<i>name</i>	The name of the command to time.
<i>state</i>	The run state to log with the timed command. See the run state table starting on page 84. You can enter 0 (MST_UNDEF) if you're uninterested in the run state.

Return Value

This function exits with one of the following results:

- TSS_OK.Success.
- TSS_NOSEVER. No previous successful call to TSSConnect ().
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

A *command* is a term or string, such as *sock* or *deposit*, that you expect to occur in client/server conversations. By placing TSSCommandStart () and TSSCommandEnd () calls around expected strings, you can record the time required to complete associated actions.

During script playback, TestManager displays progress for different virtual testers. What is displayed for a group of actions associated by TSSCommandStart () depends on the run state argument. Run states are listed in the run state table starting on page 84.

TSSCommandStart () increments IV_cmdcnt, sets the name, label, and run state for TestManager, and sets the beginning time stamp for the log entry. TSSCommandEnd () restores the TestManager run state to the run state that was in effect immediately before TSSCommandStart ().

TSSEnvironmentOp()

Example

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

```
s32 retVal = TSSCommandStart("initTimer", "Login", MST_WAITRESP);
```

See Also

TSSCommandEnd(), TSSLogCommand()

TSSEnvironmentOp()

Sets a virtual tester environment variable.

Syntax

```
s32 TSSEnvironmentOp(EvarKey envVar, EvarOp envOp, EvarValue  
*envVal)
```

Element	Description
<i>envVar</i>	The environment variable to operate on. Valid values are listed in the EvarKey enumeration.
<i>envOP</i>	The operation to perform. Valid values are listed in the EvarOp enumeration.
<i>envVal</i>	The value operated on as specified by <i>envOP</i> to produce the new value for <i>envVar</i> .

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_INVALID. The timer label is invalid, or there is no unlabeled timer to stop.
- TSS_ABORT. 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

EvarOP is defined as follows:

```
typedef enum EvarOP EvarOP;
enum EvarOP {
    EVOP_eval,
    EVOP_pop,
    EVOP_push,
    EVOP_reset,
    EVOP_restore,
    EVOP_save,
    EVOP_set,
    EVOP_END
};
```

EvarKey is defined as follows:

```
typedef enum EvarKey EvarKey;
enum EvarKey {
    EVAR_Think_avg = 0,
    EVAR_Think_sd,
    EVAR_Think_dist,
    EVAR_Think_def,
    EVAR_Think_max,
    EVAR_Think_dly_scale,
    EVAR_Think_cpu_threshold,
    EVAR_Think_cpu_dly_scale,
    EVAR_Initial_dly_max,
    EVAR_Delay_dly_scale,
    EVAR_Log_level,
    EVAR_Record_level,
    EVAR_Suspend_check,
    EVAR_LogEvent_control
    EVAR_LogData_control
    EVAR_TSSDisable
    EVAR_END
};
```

TSSGetTime()

EvarValue is defined as follows:

```
typedef union EvarValue EvarValue;
union EvarValue {
s32    envInt;
char   *envStr;
s32    envSet;
};
```

where:

- envInt is used for integer environment variables.
- envStr is used for string environment variables that may have unrestricted values.
- envSet specifies the index into a set of specific values used for string environment variables that have a predefined set of possible values.

Example

This example turns off `EVAR_Suspend_check` before the start of a block of code and then turns it back on at the end of the block.

```
s32 retVal = TSSEnvironmentOP (EVAR_Suspend_check, EVOP_push, "OFF");
/* input emulation code */
retVal = TSSEnvironmentOP (EVAR_Suspend_check, EVOP_pop, "ON");
```

TSSGetTime()

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

Syntax

```
s32 TSSGetTime(void)
```

Return Value

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

- `TSS_OK`. Success.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_ABORT`. 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 call to `TSSSession.Connect()`, or since the value of `CTXT_timeZero` set by `TSSContext()`.

Example

This example stores the elapsed time in *etime*.

```
s32 etime = TSSGetTime();
```

TSSInternalVarGet()

Gets the value of an internal variable.

Syntax

```
s32 TSSInternalVarGet(IVKey internVar, IVValue *ivVal)
```

Element	Description
<i>internVar</i>	The internal variable to operate on. Defined in the <code>IVKey</code> enum structure shown on page 52.
<i>ivVal</i>	OUTPUT. The returned value of the specified <i>internVar</i> .

Return Value

This function returns one of the following values:

- `TSS_OK`. Success.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_INVALID`. The timer label is invalid, or there is no unlabeled timer to stop.
- `TSS_ABORT`. 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.

The data type `IVKey` is defined as follows:

```
typedef enum IVKey IVKey;
enum IVKey {
    IV_fcs_ts,
    IV_lcs_ts,
    IV_fcr_ts,
    IV_lcr_ts,
    IV_lineno,
    IV_cmdcnt,
    IV_uid,
    IV_ncxmit,
    IV_ncrecv,
    IV_ncnnull,
    IV_nusers,
    IV_nkxmit,
    IV_nrows,
    IV_ncols,
    IV_row,
    IV_col,
    IV_fs_ts,
    IV_ls_ts,
    IV_fr_ts,
    IV_lr_ts,
    IV_nxmit,
    IV_nrecv,
    IV_button_no,
    IV_fuxe_ts,
    IV_luxe_ts,
    IV_uxe_cnt,
    IV_ig_fs_ts,
    IV_ig_ls_ts,
    IV_ig_eot_ts,
    IV_prev_ig_fs_ts,
    IV_prev_ig_ls_ts,
    IV_npixels_act,
    IV_npixels_exp,
    IV_npixels_diff,
    IV_xwin_diff_level,
    IV_screen,
    IV_error,
    IV_total_rows,
    IV_statement_id,
    IV_error_logs,
    IV_cursor_id,
    IV_fc_ts,
    IV_lc_ts,
```

```

    IV_total_nrecv,
    IV_error_type,
    IV_tux_tpurcode,
    IV_command,
    IV_response,
    IV_source_file,
    IV_task_file,
    IV_cmd_id,
    IV_mcommand,
    IV_alltext,
    IV_error_text,
    IV_column_headers,
    IV_total_response,
    IV_script,
    IV_version,
    IV_user_group,
    IV_host,
    IV_refURI,
    IV_END
};

```

The IVValue data type is defined as follows:

```

typedef union IVValue IVValue;
union IVValue {
s32    ivInt;
char    *ivStr;
};

```

where `ivInt` is used for integer internal variables and `ivStr` for string internal variables.

Example

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

```
s32 retVal = TSSInternalVarGet(IV_error, IVVal);
```

TSSThink()

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

Syntax

```
s32 TSSThink(s32 thinkAverage)
```

TSSTimerStart()

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

Return Value

This function exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_ABORT`. 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 `TSSEnvironmentOp()` on page 48.

Example

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

```
s32 retVal = TSSThink(0);
```

See Also

`TSSThinkTime()`

TSSTimerStart()

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

Syntax

```
s32 TSSTimerStart(char *label, s32 timeStamp)
```

Element	Description
<i>label</i>	The name of the timer to be inserted into the log. If specified as NULL, an unlabeled timer is created. Only one unlabeled timer is supported at a time.
<i>timeStamp</i>	An integer specifying a time stamp to override the current time. If specified as 0, the current time is logged.

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSERVER. No previous successful call to TSSConnect().
- TSS_ABORT. 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 TSSTimerStop(). The TestManager reporting system uses captured timing information for performance analysis reports.

Example

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

```
TSSTimerStart("event1",0);
/* actions to be timed */
TSSTimerStop("event1",0);
```

See Also

TSSTimerStop()

TSSTimerStop()

Marks the end of a block of timed actions.

Syntax

```
s32 TSSTimerStop(char *label, s32 timeStamp, u32 rmFlag)
```

Element	Description
<i>label</i>	The name of the timer to be stopped and logged. If <i>label</i> does not match a label entered with a previous TSSTimerStart() call, the most recent unlabeled timer is stopped.
<i>time stamp</i>	If specified as 0, the current time is recorded.
<i>rmFlag</i>	Specify as 0 to stop the timer without removing it; otherwise, specify as nonzero. A timer that is not removed can be stopped multiple times in order to measure intervals comprising this timed event.

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSERVER. No previous successful call to TSSConnect().
- TSS_INVALID. The timer label is invalid, or there is no unlabeled timer to stop.
- TSS_ABORT. 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 TSSTimerStart(). If the specified *label* was not set by a previous TSSTimerStart() but an unlabeled timer exists, this call uses the start time specified with TSSTimerStart() for the unlabeled timer. If *rmFlag* is specified as 0, multiple invocations of TSSTimerStop() are allowed against a single TSSTimerStart(). 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.

```
TSSTimerStart(NULL, 0);
/* actions to be timed */
TSSTimerStop("event1", 0, 0);
/* other actions to be timed */
TSSTimerStop("event2", 0, 0);
```

See Also

TSSTimerStart()

Utility Services

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

Summary

The following table lists the utility functions.

Function	Description
TSSApplicationPid()	Gets the process ID of an application.
TSSApplicationStart()	Starts an application.
TSSApplicationWait()	Waits for an application to terminate.
TSSDelay()	Delays the specified number of milliseconds.
TSSErrorDetail()	Retrieves error information about a failure.
TSSGetComputerConfigurationAttributeList()	Gets the list of computer configuration attributes and their values.
TSSGetComputerConfigurationAttributeValue()	Gets the value of a computer configuration attribute.
TSSGetPath()	Gets a pathname.
TSSGetScriptOption()	Gets the value of a script playback option.
TSSGetTestCaseConfigurationAttribute()	Gets the value of a test case configuration attribute.
TSSGetTestCaseConfigurationAttributeList()	Gets the list of test case configuration attributes and their values.
TSSGetTestCaseConfigurationName()	Gets the name of the configuration (if any) associated with the current test case.
TSSGetTestCaseName()	Gets the name of the test case in use.
TSSGetTestToolOption()	Gets a test case tool option.
TSSJavaApplicationStart()	Starts a Java application.
TSSNegExp()	Gets the next negative exponentially distributed random number with the specified mean.

Function	Description
TSSRand ()	Gets the next random number.
TSSSeedRand ()	Seeds the random number generator.
TSSStdErrPrint ()	Prints a message to the virtual tester's error file.
TSSStdOutPrint ()	Prints a message to the virtual tester's output file.
TSSUniform ()	Gets the next uniformly distributed random number in the specified range.
TSSUniqueString ()	Returns a unique text string.

TSSApplicationPid()

Gets the process ID of an application.

Syntax

```
s32 TSSApplicationPid(TSSAppHandle appHandle)
```

Element	Description
<i>appHandle</i>	The ID of the application whose PID you want to get. Returned by TSSApplicationStart () or TSSJavaApplicationStart ().

Return Value

On success, this function returns the system process ID of the specified application. On failure, it returns 0: call TSSErrorDetail () for information.

Comments

This function works for applications started by TSSApplicationStart () or TSSJavaApplicationStart ().

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.

```
TSSAppHandle myAppHandle = TSSApplicationStart("myAPP", "d:\myDir",
0);
s32 myAppPID = TSSApplicationPid (myAppHandle);
```

See Also

TSSApplicationStart(), TSSApplicationWait(), TSSJavaApplicationStart()

TSSApplicationStart()

Starts an application.

Syntax

```
TSSAppHandle TSSApplicationStart(char *appHandle, char
*workingDir, u32 flags)
```

Element	Description
<i>appHandle</i>	The pathname of the application to be started, which can include options and arguments. The file suffix can be omitted.
<i>workingDir</i>	The directory in which to start the application. The current directory if specified as "".
<i>flags</i>	Reserved for future use. Specify as 0.

Return Value

On success, this function returns a handle for the started application. On failure, it returns 0: call TSSErrorDetail() for information.

Comments

TSSAppHandle is defined as: typedef void *TSSAPPHandle.

Example

This example starts application myApp.

```
TSSAppHandle myAppHandle = TSSApplicationStart("myAPP", "d:\myDir",
0);
```

TSSApplicationWait()

See Also

TSSApplicationPid(), TSSApplicationWait(), TSSJavaApplicationStart()

TSSApplicationWait()

Waits for an application to terminate.

Syntax

```
s32 TSSApplicationWait(TSSAppHandle app, s32 *exitStatus, s32
    timeout)
```

Element	Description
<i>app</i>	The application that you are waiting for. Returned by TSSApplicationStart() or TSSJavaApplicationStart().
<i>exitStatus</i>	OUTPUT. If not NULL, the exit status of <i>app</i> .
<i>timeout</i>	The number of milliseconds to wait for <i>app</i> to terminate or 0 to return immediately.

Return Value

This function exits with one of the following results:

- TSS_OK.Success.
- TSS_FAIL. The application was still running when the time-out expired.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_SYSERROR. The system returned an error: call TSSErrorDetail() for information.
- TSS_NOTFOUND. The process indicated by *app* was not found. It may have terminated before this call or *app* may be an invalid handle.
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

This function works for applications started by TSSApplicationStart() or TSSJavaApplicationStart().

If *app* is still running at the time this call returns, *exitStatus* contains NULL. If *app* has terminated at the time of return, *exitStatus* contains its termination code.

Example

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

```
s32 termStatus;
TSSAppHandle myAppHandle = TSSApplicationStart("myAPP", "d:\myDir",
0);
s32 retval = TSSApplicationWait (myAppHandle, termStatus, 600);
```

See Also

`TSSApplicationPid()`, `TSSApplicationStart()`, `TSSJavaApplicationStart()`

TSSDelay()

Delays script execution for the specified number of milliseconds.

Syntax

```
s32 TSSDelay(s32 msec)
```

Element	Description
<i>msec</i>	The number of milliseconds to delay script execution.

Return Value

This function exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

Comments

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

TSSErrorDetail()

Example

This example delays execution for 10 milliseconds.

```
s32 retVal = TSSDelay(10);
```

TSSErrorDetail()

Retrieves error information about a failure.

Syntax

```
s32 TSSErrorDetail(char *errorText, s32 *len)
```

Element	Description
<i>errorText</i>	OUTPUT. Returned explanatory error message about the previous TSS call, or an empty string ("") if the previous TSS call did not fail.
<i>len</i>	The length of string <i>errorText</i> .

Return Value

This function returns `TSS_OK` if the previous call succeeded. If the previous call failed, `TSSErrorDetail()` returns one of the error codes listed below and corresponding *errorText*.

- `TSS_OK`. Success.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

Comments

If the message is too long to fit in *errorText*, it is truncated to *len* and *len* is updated to the message length.

Example

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

```
char message[256];
s32 dpid, ecode, msglen = 256;
dpid = TSSDatapoolOpen ("custdata", 0, 0, NULL);
if (dpid < 0)
{
    /* open failed, report error */
    ecode = TSSErrorDetail(message, &msglen);
    fprintf(stderr, "TSSDatapoolOpen failed. code: %d, message: %s\n",
ecode, message);
}
```

TSSGetComputerConfigurationAttributeList()

Gets the list of computer configuration attributes and their values.

Syntax

```
s32 TSSGetComputerConfigurationAttributeList (NamedValue []
**config, s32 *count)
```

Element	Description
<i>config</i>	OUTPUT. An array containing configuration name/value pairs, where <code>config[n].name</code> is the attribute name and <code>config[n].value</code> is its value.
<i>count</i>	OUTPUT. The number of rows in the <i>config</i> array.

Return Value

On success, this function returns an array of computer configuration attribute names and their values. It exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSERVER. No previous successful call to TSSConnect ().
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

TSSGetComputerConfigurationAttributeValue()

Comments

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

The pointer to *config* is valid until the next call of this function. The `NamedValue` data type is defined as follows:

```
typedef struct {
    char *Name;
    char *Value;
} NamedValue;
```

Example

This example returns the current computer configuration attribute list.

```
s32 npairs;
NamedValue *config;
s32 retVal = TSSGetComputerConfigurationAttributeList (&config,
&npairs);
```

See Also

TSSGetComputerConfigurationAttributeValue()

TSSGetComputerConfigurationAttributeValue()

Gets the value of computer configuration attribute.

Syntax

```
char *TSSGetComputerConfigurationAttributeValue (char *name)
```

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

Return Value

On success, this function returns a handle for the started application. On failure, it returns NULL: call `TSSErrorDetail()` for information.

Example

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

```
char *OSVal = TSSGetComputerConfigurationAttributeValue("Operating
System");
```

See Also

`TSSGetComputerConfigurationAttributeList()`

TSSGetPath()

Gets the pathname of a test asset.

Syntax

```
char *TSSGetPath (u32 pathKey)
```

Element	Description
<i>pathKey</i>	<p>Specifies one of these values:</p> <ul style="list-style-type: none"> ▪ <code>TSS_SOURCE_PATH</code> to get the location of the source file for the currently executing test script. On an agent, this is the root destination to which files are copied from the local computer. ▪ <code>TSS_ATTACHED_LOG_FILE_PATH</code> to get the location of files attached to the log.

Return Value

On success, this function returns the pathname of the currently executing test script. On failure, it returns NULL: call `TSSErrorDetail()` for information.

Example

This example returns the path of the currently executing test script.

```
char *scriptPath = TSSGetPath(TSS_SOURCE_PATH);
```

See Also

`TSSUniqueString()`

TSSGetScriptOption()

Gets the value of a test script playback option.

Syntax

```
char *TSSGetScriptOption(char *optionName)
```

Element	Description
<i>optionName</i>	The name of the script option whose value is returned.

Return Value

On success, this function returns the value of the specified script option. The function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

The Test Script Execution Adapter (TSEA) for a test script type might support session or task execution options. If an adapter does support options, you can get their values with this call. For more information, see *Communication Overview* on page 15.

The returned pointer to *optVal* is valid until the next TSSGetScriptOption() call.

Example

This example gets the value of the script option `repeat_count`.

```
char *optVal;
if (optVal = TSSGetScriptOption("repeat_count"))
    printf("The value of repeat_count is %s\n", repeat_count);
```

See Also

SessionSetOption(), *TaskSetOption()*

TSSGetTestCaseConfigurationAttribute()

Gets the value of the specified test case configuration attribute.

Syntax

```
s32 *TSSGetTestCaseConfigurationAttribute (char *name,
    TestCaseConfigurationAttribute *config)
```

Element	Description
<i>name</i>	Specifies the name of the configuration attribute to be returned.
<i>config</i>	OUTPUT. The returned test case configuration value.

Return Value

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

- TSS_OK.Success.
- TSS_NOSEVER. No previous successful call to TSSConnect ().
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

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

The `TestCaseConfigurationAttribute` data type is defined as follows:

```
typedef struct {
    char *name;
    char *operator
    char *value;
} TestCaseConfigurationAttribute;
```

Example

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

```
TestCaseConfigurationAttribute OSVal =
TSSGetTestCaseConfigurationAttribute("Operating System");
```

See Also

TSSGetTestCaseConfigurationAttributeList()

TSSGetTestCaseConfigurationAttributeList()

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

Syntax

```
s32 *TSSGetTestCaseConfigurationAttributeList
    (TestCaseConfigurationAttribute **config, s32 *count)
```

Element	Description
<i>config</i>	OUTPUT. An array containing configuration name/operator/value triplets, where <code>config[n].name</code> is the attribute name, <code>config[n].operator</code> is the operator, and <code>config[n].value</code> is the attribute value.
<i>count</i>	OUTPUT. The number of rows in the <i>config</i> array.

Return Value

On success, this function returns an array of test case configuration attribute names, base values, and operators. It exits with one of the following results:

- TSS_OK.Success.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

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

The `TestCaseConfigurationAttribute` data type is defined as follows:

```
typedef struct {
    char *name;
    char *operator;
    char *value;
} TestCaseConfigurationAttribute;
```

Example

This example returns the current test case configuration attribute list.

```
s32 nRows;
TestCaseConfigurationAttribute *config;
s32 retVal = TSSGetTestCaseConfigurationAttributeList (&config,
&nRows);
```

See Also

`TSSGetTestCaseConfigurationAttribute()`

TSSGetTestCaseConfigurationName()

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

Syntax

```
char *TSSGetTestCaseConfigurationName(void)
```

Return Value

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

- `TSS_OK.Success`.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_ABORT`. 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.

```
char *tcConfig = TSSGetTestCaseConfigurationName();
```

TSSGetTestCaseName()

Gets the name of the test case in use.

Syntax

```
char *TSSGetTestCaseName(void)
```

Return Value

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

- TSS_OK. Success.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_ABORT. 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.

The returned pointer to *testcase* is valid until the next TSSGetTestCaseName() call.

Example

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

```
char *tcName;
if (tcName = TSSGetTestCaseName())
    printf("The test case is %s\n", tcName);
```

TSSGetTestToolOption()

Gets the value of a test tool execution option.

Syntax

```
char *TSSGetTestToolOption(char *optionName)
```

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

Return Value

On success, this function returns the value of the specified test tool execution option. On failure, it returns NULL: call `TSSErrorDetail()` for information.

Comments

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

Example

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

```
char *optval = TSSGetTestToolOption ("persist");
```

TSSJavaApplicationStart()

Starts a Java application.

Syntax

```
TSSAppHandle TSSJavaApplicationStart(char *app, char
    *workingDir, char *classPath, char *JVM, char *JVMOptions)
```

Element	Description
<i>app</i>	The pathname of the application to be started, which can include options and arguments. The file suffix can be omitted.
<i>workingDir</i>	The directory in which to start the application. The current directory if specified as NULL.
<i>classPath</i>	The Java CLASSPATH or NULL. The specified value replaces the current CLASSPATH.
<i>JVM</i>	The pathname of Java Virtual Machine. If specified as NULL, <code>java.exe</code> is used on Windows machines and <code>java</code> on UNIX agent platforms.
<i>JVMOptions</i>	Any valid JVM options or NULL.

Return Value

On success, this function returns a handle for the started application. On failure, it returns NULL: call `TSSErrorDetail()` for information.

TSSNegExp()

Comments

TSSAppHandle is defined as: `typedef void *TSSAPPHandle`.

Example

This example starts application myJavaApp.

```
TSSAppHandle myAppHandle = TSSJavaApplicationStart("myJavaAPP", "",
"", "", "");
```

See Also

`TSSApplicationPid()`, `TSSApplicationStart()`, `TSSApplicationWait()`

TSSNegExp()

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

Syntax

```
s32 TSSNegExp(s32 mean)
```

Element	Description
<i>mean</i>	The mean value for the distribution.

Return Value

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

- `TSS_OK`. Success.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_ABORT`. 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 mean, then `TSSErrorDetail()` returns `TSS_OK`.

Example

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

```
s32 retVal = TSSSeedRand(10);
s32 next = TSSNegExp(10);
```

See Also

`TSSRand()`, `TSSSeedRand()`, `TSSUniform()`

TSSRand()

Gets the next random number.

Syntax

```
s32 TSSRand(void)
```

Return Value

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

- `TSS_OK`. Success.
- `TSS_NOSERVER`. No previous successful call to `TSSConnect()`.
- `TSS_ABORT`. 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.

TSSSeedRand()

Example

This example gets the next random number.

```
s32 next = TSSRand();
```

See Also

TSSSeedRand(), TSSNegExp(), TSSUniform()

TSSSeedRand()

Seeds the random number generator.

Syntax

```
s32 TSSSeedRand(u32 seed)
```

Element	Description
<i>seed</i>	The base integer.

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_ABORT. 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.

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

Example

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

```
s32 retVal = TSSSeedRand(10);
```

See Also

TSSRand(), TSSNegExp(), TSSUniform()

TSSePrint()

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

Syntax

```
s32 TSSePrint(char *message)
```

Element	Description
<i>message</i>	The string to print.

Return Value

This function exits with one of the following results:

- TSS_OK.Success.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Example

This example prints to the error file the message Login failed.

```
s32 retVal = TSSePrint("Login failed");
```

See Also

TSSPrint()

TSSPrint()

TSSPrint()

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

Syntax

```
s32 TSSPrint(char *message)
```

Element	Description
<i>message</i>	The string to print.

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Example

This example prints the message `Login successful`.

```
s32 retVal = TSSPrint("Login successful");
```

See Also

TSSePrint()

TSSUniform()

Gets the next uniformly distributed random number.

Syntax

```
s32 TSSUniform(s32 low, s32 high)
```

Element	Description
<i>low</i>	The low end of the range.
<i>high</i>	The high end of the range.

Return Value

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

- `TSS_OK`. Success.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_ABORT`. 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 `TSS_OK`.

Example

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

```
int next = TSSUniform(-10,10);
```

See Also

`TSSRand()`, `TSSSeedRand()`, `TSSNegExp()`

TSSUniqueString()

Returns a unique text string.

Syntax

```
char *TSSUniqueString(void)
```

Return Value

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

Comments

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

Example

This example returns a unique text string.

```
char *str = TSSUniqueString();
```

Monitor Services

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 functions support the TestManager monitoring options.

Summary

The following table lists the monitoring functions.

Function	Description
TSSDisplay()	Sets a message to be displayed by the monitor.
TSSPositionGet()	Gets the script source file name or line number position.
TSSPositionSet()	Sets the script source file name or line number position.
TSSReportCommandStatus()	Gets the runtime status of a command.
TSSRunStateGet()	Gets the run state.
TSSRunStateSet()	Sets the run state.

TSSDisplay()

Sets a message to be displayed by the monitor.

Syntax

```
s32 TSSDisplay(char *message)
```

Element	Description
<i>message</i>	The message to be displayed by the progress monitor.

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOOP. The TSS server is running proxy.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

This message is displayed until overwritten by another call to TSSDisplay().

Example

This example sets the monitor display to Beginning transaction.

```
s32 retVal = TSSDisplay("Beginning transaction");
```

TSSPositionGet()

Gets the test script file name or line number position.

Syntax

```
s32 TSSPositionGet(char **srcFile, u32 *lineNumber)
```

Element	Description
<i>srcFile</i>	OUTPUT. The name of a source file. After a successful call, this variable contains the name of the source file that was specified with the most recent <code>TSSPositionSet()</code> call.
<i>lineNumber</i>	OUTPUT. The name of a local variable. After a successful call, this variable contains the current line position in <i>srcFile</i> .

Return Value

On success, this function returns *srcFile* and *lineNumber* as explained in the preceding table. The function exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_ABORT`. 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. `TSSPositionSet()` and `TSSPositionGet()` 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 `TSSPositionSet()`. 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.

```
char ** scriptFile;
u32 *lineNumber;
s32 retVal = TSSPositionGet(scriptFile, lineNumber);
```

See Also

`TSSPositionSet()`

TSSPositionSet()

Sets the test script file name or line number position.

Syntax

```
s32 TSSPositionSet(char *srcFile, u32 lineNumber)
```

Element	Description
<i>srcFile</i>	The name of the test script, or NULL for the current test script.
<i>lineNumber</i>	The number of the line in <i>srcFile</i> to set the cursor to, or 0 for the current line.

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_ABORT. 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. TSSPositionSet() and TSSPositionGet() 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.

```
s32 retVal = TSSPositionSet("checkLogin",0);
```

See Also

TSSPositionSet()

TSSReportCommandStatus()

Reports the runtime status of a command.

Syntax

```
s32 TSSReportCommandStatus (s32 status)
```

Element	Description
<i>status</i>	The status of a command. Can be one of the following: <ul style="list-style-type: none"> ▪ TSS_CMD_STAT_FAIL ▪ TSS_CMD_STAT_PASS ▪ TSS_CMD_STAT_WARN ▪ TSS_CMD_STAT_INFO

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOOP. The TSS server is running proxy.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_INVALID. The entered *status* is invalid.
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Example

This example reports a failure command status.

```
s32 retVal = TSSReportCommandStatus (TSS_CMD_STAT_FAIL);
```


TSSRunStateGet()

Gets the run state.

Syntax

```
s32 TSSRunStateGet(void)
```

Return Value

On success, this function returns one of the run state values listed in the run state table starting on page 84. The function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_ABORT. 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.

```
s32 orig = TSSRunStateGet();
```

See Also

TSSRunStateSet()

TSSRunStateSet()

Sets the run state.

Syntax

```
s32 TSSRunStateSet(RunState state)
```

Element	Description
<i>state</i>	The run state to set. Enter one of the run state values listed in the run state table starting on page 84.

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSEVER. No previous successful call to TSSConnect ().
- TSS_INVALID. Invalid run state.
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

RunState is defined as follows:

```
typedef u32 RunState;
```

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
MST_BIND	iiop_bind in progress
MST_BUTTON	X button action
MST_CLEANUP	cleaning up
MST_CPUDLY	cpu delay
MST_DELAY	user-requested delay
MST_DSPLYRESP	displaying response
MST_EXITED	exited
MST_EXITSQABASIC	exited SQA Basic code
MST_EXTERN_C	executing external C code

Run State	Meaning
MST_FIND	find_text find_point
MST_GETTASK	waiting for task assignment
MST_HTTPCONN	waiting for http connection
MST_HTTPDISC	waiting for http disconnect
MST_IIOPIVOKE	iiop_invoke in progress
MST_INCL	mask including above basic states
MST_INIT	doing startup initialization
MST_INITTASK	initializing task
MST_ITDLY	intertask delay
MST_MOTION	X motion
MST_PMATCH	matching response (precv)
MST_RECV_DELAY	line_speed delay in recv
MST_SATEXEC	executing satellite script
MST_SEND	httpsocket send
MST_SEND_DELAY	line_speed delay in send
MST_SHVBLCK	blocked from shv access
MST_SHVREAD	V_VP: reading shared variable
MST_SHVWAIT	user requested shv wait
MST_SOCKCONN	waiting for socket connection
MST_SOCKDISC	waiting for socket disconnect
MST_SQABASIC_CODE	running SQABasic code
MST_SQLCONN	waiting for SQL client connection
MST_SQLDISC	waiting for SQL client disconnect
MST_SQLEXEC	executing SQL statements
MST_STARTAPP	SQABasic: starting app
MST_SUSPENDED	suspended
MST_TEST	test case, emulate

Run State	Meaning
MST_THINK	thinking
MST_TRN_PACING	transactor pacing delay
MST_TUXEDO	Tuxedo execution
MST_TYPE	typing
MST_UNDEF	user's micro_state is undefined
MST_USERCODE	SQAVu user code
MST_WAITOBJ	SQABasic: waiting for object
MST_WAITRESP	waiting for response
MST_WATCH	interactive -W watch record
MST_XCLNTCONN	waiting for http connection
MST_XCLNTCONN	waiting for socket connection
MST_XCLNTCONN	waiting for SQL client connection
MST_XCLNTCONN	waiting for X client connection
MST_XCLNTDISC	waiting for http disconnect
MST_XCLNTDISC	waiting for socket disconnect
MST_XCLNTDISC	waiting for SQL client disconnect
MST_XCLNTDISC	waiting for X client disconnect
MST_XMOVEWIN	X move window
MST_XQUERY	X query function
MST_XSYNC	X sync state during X query
MST_XWINCMP	xwindow_diff comparing windows
MST_XWINDUMP	xwindow_diff dumping window
N_MST_INCL	number of above states

Example

This example sets the run state to `MST_WAITRESP`.

```
s32 retVal = TSSRunStateSet(MST_WAITRESP);
```

See Also

TSSRunStateGet ()

Synchronization Services

Use the synchronization functions 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 functions.

Function	Description
TSSSharedVarAssign()	Performs a shared variable assignment operation.
TSSSharedVarEval()	Gets the value of a shared variable and operates on the value as specified.
TSSSharedVarWait()	Waits for the value of a shared variable to match a specified range.
TSSSyncPoint()	Puts a synchronization point in a script.

TSSSharedVarAssign()

Performs a shared variable assignment operation.

Syntax

```
s32 TSSSharedVarAssign(char *name, s32 value, ShVarOp op, s32
    *returnVal)
```

Element	Description
<i>name</i>	The name of the shared variable to operate on.
<i>value</i>	The right-side value of the assignment expression.

Element	Description
<i>op</i>	Assignment operator. Can be one of the following: <ul style="list-style-type: none"> ▪ SHVOP_assign ▪ SHVOP_add ▪ SHVOP_subtract ▪ SHVOP_multiply ▪ SHVOP_divide ▪ SHVOP_modulo ▪ SHVOP_and ▪ SHVOP_or ▪ SHVOP_xor ▪ SHVOP_shiftleft ▪ SHVOP_shiftright
<i>returnVal</i>	OUTPUT. If not specified as NULL, the resulting value of <i>name</i> after application of <i>op</i> value.

Return Value

On success, this function retrieves the value of the specified shared variable before and after it has been operated on. The function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSERVER. No previous successful call to TSSConnect ().
- TSS_INVALID. The entered *name* is not a shared variable.
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

The data type ShVarOp is defined as follows:

```
typedef enum ShVarOp ShVarOp;
enum ShVarOp {
    SHVOP_assign,
    SHVOP_add,
    SHVOP_subtract,
    SHVOP_multiply,
    SHVOP_divide
    SHVOP_modulo,
    SHVOP_and,
    SHVOP_or,
    SHVOP_xor,
    SHVOP_shiftleft
```

```

    SHVOP_shiftright
    SHVOP_END
}

```

TSSSharedVarAssign("myVar", 5, SHVOP_add, NULL) is equivalent to myVar += 5.

Example

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

```

s32 returnval = 5;
s32 retVal = TSSSharedVarAssign("lineCounter", val, SHVOP_add,
returnVal);

```

See Also

TSSSharedVarEval(), TSSSharedVarWait()

TSSSharedVarEval()

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

Syntax

```
s32 TSSSharedVarEval(char *name, s32 *value, ShVarAdj op)
```

Element	Description
<i>name</i>	The name of the shared variable to operate on.
<i>value</i>	OUTPUT. A local container into which the value of <i>name</i> is retrieved.
<i>op</i>	Increment/decrement operator for the returned value: Can be one of the following: <ul style="list-style-type: none"> ▪ SHVADJ_none SHVADJ_pre_inc ▪ SHVADJ_post_inc ▪ SHVADJ_pre_dec ▪ SHVADJ_post_dec

TSSSharedVarWait()

Return Value

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

- TSS_OK.Success.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_INVALID. The entered *name* is not a shared variable.
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

The data type ShVarAdj is defined as follows:

```
typedef enum ShVarAdj ShVarAdj;  
enum ShVarAdj {  
    SHVADJ_none,  
    SHVADJ_pre_inc,  
    SHVADJ_post_inc,  
    SHVADJ_pre_dec,  
    SHVADJ_post_dec  
}
```

Example

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

```
s32 val;  
s32 retVal = TSSSharedVarEval("lineCounter", val, SHVADJ_post_dec);
```

See Also

TSSSharedVarAssign(), TSSSharedVarWait()

TSSSharedVarWait()

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

Syntax

```
s32 TSSSharedVarWait(char *name, s32 min, s32 max, s32 adjust,  
    s32 timeout, s32 *returnVal)
```


Element	Description
<i>name</i>	The name of the shared variable to operate on.
<i>min</i>	The low range for the value of <i>name</i> .
<i>max</i>	The high range for the value of <i>name</i> .
<i>adjust</i>	The value to increment/decrement the named shared variable by once it meets the <i>min</i> – <i>max</i> range.
<i>timeout</i>	The time-out preference (how long to wait for the condition to be met). Enter one of the following: <ul style="list-style-type: none"> ▪ 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 <i>name</i> to meet the criteria, before timing out with and returning an exit value of 1 (met) or 0 (not met).
<i>returnVal</i>	OUTPUT. The value of <i>name</i> at the time of the return, before any possible adjustment. If <i>timeout</i> expired before the return, the value is not adjusted. Otherwise, <i>returnVal</i> is incremented/decremented by <i>adjust</i> .

Return Value

On success, this function returns 1 (condition was met before time-out) or 0 (time-out expired before the condition was met). The function exits with one of the following results:

- TSS_NOSEVER. No previous successful call to TSSConnect ().
- TSS_INVALID. The entered *name* is not a shared variable.
- TSS_ABORT. 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 `TSSSharedVarWait()` — 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 `TSSSharedVarWait()` 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 `TSSSharedVarWait()` 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.)

```
s32 svVal = 0;
s32 retVal = TSSSharedVarWait("inProgress",10,20,0,60000,svVal);
```

See Also

`TSSSharedVarAssign()`, `TSSSharedVarEval()`

TSSSyncPoint()

Puts a synchronization point in a script.

Syntax

```
s32 TSSSyncPoint(char *label)
```

Element	Description
<i>label</i>	The name of the synchronization point.

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOOP. The TSS server is running proxy.
- TSS_NOSEVER. No previous successful call to TSSConnect().
- TSS_INVALID. The synchronization point *label* is invalid.
- TSS_ABORT. 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 TSSyncPoint() 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 TSSyncPoint(), 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.

```
s32 retVal = TSSyncPoint("BlockUntilSaveComplete");
```

Session Services

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.

In a given suite run, a session can be run directly (inside the TestManager process space) or by a separate TSS server process (proxy). The latter happens only if the following two conditions are met:

- The test script(s) is executed by a stand-alone process (outside of TestManager) and is linked with the link library `rtssremote.lib`.
- The first script of a given type in a suite that can be executed by a TSS proxy server calls `TSSServerStart()`.

Summary

Applications can use the session functions listed in the following table to manage proxy TSS servers and sessions on behalf of test scripts. These functions are not needed for sessions that are directly executed by TestManager.

Function	Description
<code>TSSConnect()</code>	Connects to a TSS proxy server.
<code>TSSContext()</code>	Passes context information to a TSS server.
<code>TSSDisconnect()</code>	Disconnects from a TSS proxy server.
<code>TSSServerStart()</code>	Starts a TSS proxy server.
<code>TSSServerStop()</code>	Stops a TSS proxy server.
<code>TSSShutdown()</code>	Stops logging and initializes TSS.

TSSConnect()

Connects to a TSS proxy server.

Syntax

```
s32 TSSConnect (char *host, u16 port, s32 id)
```

Element	Description
<i>host</i>	The name (or IP address in quad dot notation) of the host on which the proxy TSS server process is running.
<i>port</i>	The listening port for the TSS server on <i>host</i> , or 0 (recommended) to let TestManager select the port.
<i>id</i>	The connection identifier.

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOOP. A connection and ID had already been established for this execution thread.
- TSS_NOSERVER. No TSS server was listening on *port*.
- TSS_SYSERROR. A system error occurred. Call `TSSErrorDetail()` for information.
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

For scripts that are executed by a proxy process rather than directly by the TSEE, this function must be called before any other TSS functions. This function is also required when a script (or TSEA) starts a new thread of execution.

The proxy TSS DLL uses *host* and *port* (the host and port parameters passed to `SessionOpen()` in the TSEA) to establish a connection with the correct TSEE.

The direct TSS DLL ignores *host* and *port*, and associates the *id* with the current execution thread. If the thread already had an ID, *id* is ignored. (You cannot change *id*.)

TSSContext()

Example

This example connects to a TSS server running on host 192.36.25.107. The *port* is defined in the example for TSSServerStart ().

```
s32 retVal = TSSConnect ("192.36.25.107",port,0);
```

See Also

TSSServerStart ()

TSSContext()

Passes context information to a TSS server.

Syntax

```
s32 TSSContext (ContextKey ctx, void *value)
```

Element	Description
<i>ctx</i>	The type of context information to pass: Can be one of the following: <ul style="list-style-type: none">▪ CTXT_workingDir▪ CTXT_datapoolDir▪ CTXT_timeZero▪ CTXT_todZero▪ CTXT_logDir▪ CTXT_logFile▪ CTXT_logData▪ CTXT_testScript▪ CTXT_style▪ CTXT_sourceUID
<i>value</i>	The information of type <i>ctx</i> to pass.

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSERVER. No previous successful call to TSSConnect ().
- TSS_INVALID. The specified *ctx* is invalid.
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

This call is useful for test scripts that are executed by a stand-alone process — outside the TestManager framework — and that also make TSS calls. The call passes information, such as the log file name, that would be passed through shared memory if the script were executed by TestManager.

Test scripts that are executed by a proxy TSS server process should make this call immediately after TSSConnect (), before accessing any other TSS services. Otherwise, inconsistent results can occur.

ContextKey is defined as follows:

```
enum ContextKey {
    CTXT_workingDir,
    CTXT_datapoolDir,
    CTXT_timeZero,
    CTXT_todZero,
    CTXT_logDir
    CTXT_logFile
    CTXT_logData
    CTXT_testScript
    CTXT_style
    CTXT_sourceUID
    CTXT_END
};
typedef enum ContextKey ContextKey;
```

Example

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

```
s32 retVal = TSSContext(CTXT_workingDir, "C:\\temp");
```

TSSDisconnect()

TSSDisconnect()

Disconnects from a TSS proxy server.

Syntax

```
void TSSDisconnect (void)
```

Return Value

None.

Comments

This call closes the connection established by `TSSConnect()` and performs any required cleanup operations.

Example

This example disconnects from the TSS server.

```
TSSDisconnect();
```

TSSServerStart()

Starts a TSS proxy server.

Syntax

```
s32 TSSServerStart (u16 *port)
```

Element	Description
<i>port</i>	The listening port for the TSS server. If specified as 0 (recommended), the system chooses the port and returns its number to <i>port</i> .

Return Value

This function exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_NOOP`. A TSS server was already listening on *port*.
- `TSS_NOSEVER`. Start failure. Call `TSSErrorDetail()` for information.

- `TSS_SYSERROR`. A system error occurred. Call `TSSErrorDetail()` for information.
- `TSS_ABORT`. 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`.

```
u16 *port = 0;
s32 retVal = TSSServerStart (&port);
```

See Also

`TSSServerStop()`

TSSServerStop()

Stops a TSS proxy server.

Syntax

```
s32 TSSServerStop (u16 port)
```

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

TSSShutdown()

Return Value

This function exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_NOOP`. No TSS server was listening on *port*.
- `TSS_INVALID`. No proxy TSS server was found or stopped.
- `TSS_SYSERROR`. A system error occurred. Call `TSSErrorDetail()` for information.
- `TSS_ABORT`. 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 that was started by the example for `TSSServerStart()`.

```
s32 retval = TSSServerStop (port);
```

See Also

`TSSServerStart()`

TSSShutdown()

Stops logging and initializes TSS.

Syntax

```
s32 TSSShutdown (void)
```

Return Value

This function exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_NOSEVER`. No previous successful call to `TSSConnect()`.
- `TSS_INVALID`. The specified *ctx* is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

Comments

This call stops logging functions, pauses a playback session, and initializes TSS to resume logging and executing the next task.

Example

This example shuts down logging during session execution so that logging can be restarted for the next task.

```
s32 retval = TSSShutdown ();
```

Advanced Services

You can use the advanced functions 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. As a result, the functions need not and usually should not be performed by individual test scripts.

Summary

The following table lists the advanced functions.

Function	Description
TSSInternalVarSet ()	Sets the value of an internal variable.
TSSLogCommand ()	Logs a command event.
TSSThinkTime ()	Calculates a think-time average.

TSSInternalVarSet()

Sets the value of an internal variable.

Syntax

```
s32 TSSInternalVarSet (IVKey internVar, IVValue ivVal)
```

Element	Description
<i>internVar</i>	The internal variable to operate on. Internal variables and their values are listed in the table starting on page 52. See page 52 for the IVKey and page 53 for the IVValue definitions.
<i>ivVal</i>	The new value for <i>internVar</i> .

Return Value

The function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSEVER. No previous successful call to TSSConnect () .
- TSS_INVALID. The timer label is invalid, or there is no unlabeled timer to stop.
- TSS_ABORT. 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 IV_cmdcnt to 0.

```
s32 retVal = TSSInternalVarSet (IV_cmdcnt, 0);
```

See Also

TSSInternalVarGet ()

TSSLogCommand()

Logs a command event.

Syntax

```
s32 TSSLogCommand(char *name, char *label, s16 result, char
    *description, s32 starttime, s32 endtime, char *logdata, s32
    propertyCount, NamedValue *property)
```

Element	Description
<i>name</i>	The command name.
<i>label</i>	The event label.
<i>result</i>	Specifies the notification preference regarding the result of the call. Can be one of the following: <ul style="list-style-type: none"> ▪ TSS_LOG_RESULT_NONE (default: no notification) ▪ TSS_LOG_RESULT_PASS ▪ TSS_LOG_RESULT_FAIL ▪ TSS_LOG_RESULT_WARN ▪ TSS_LOG_RESULT_STOPPED ▪ TSS_LOG_RESULT_INFO ▪ TSS_LOG_RESULT_COMPLETED ▪ TSS_LOG_RESULT_UNEVALUATED 0 specifies the default.
<i>description</i>	Contains the string to be displayed in the event of failure.
<i>starttime</i>	An integer indicating a time stamp. If specified as 0, the logged time stamp is the later of the values contained in internal variables IV_fcs_ts and IV_fcr_ts.
<i>endtime</i>	An integer indicating a time stamp. If specified as 0, the time set by TSSCommandEnd is logged.
<i>logdata</i>	Text to be logged describing the ended command.
<i>propertyCount</i>	Specifies the number of rows in the property array.
<i>property</i>	An array containing property name/value pairs, where <code>property [n] . name</code> is the property name and <code>property [n] . value</code> is its value.

Return Value

This function exits with one of the following results:

- TSS_OK.Success.
- TSS_NOSERVER.No previous successful call to TSSConnect () .
- TSS_ABORT.Pending abort resulting from a user request to stop a suite run.

TSSThinkTime()

Comments

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

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

An event and any data associated with it are logged only if the specified *result* preference matches associated settings in the `EVAR_LogData_control` or `EVAR_LogEvent_control` environment variables. Alternatively, the logging preference may be set with the `EVAR_Log_level` and `EVAR_Record_level` environment variables. The `TSS_LOG_RESULT_STOPPED`, `TSS_LOG_RESULT_COMPLETED`, and `TSS_LOG_RESULT_UNEVALUATED` preferences are intended for internal use.

`NamedValue` is defined as follows:

```
typedef struct {
    char *Name;
    char *Value;
} NamedValue;
```

Example

This example logs a message for a login script.

```
s32 retVal = TSSLogCommand("Login", "initTimer",
TSS_LOG_RESULT_PASS, "Command timer failed", 0, 0, "Login command
completed", NULL);
```

See Also

`TSSCommandStart()`, `TSSCommandEnd()`

TSSThinkTime()

Calculates a think-time average.

Syntax

```
s32 TSSThinkTime(s32 thinkAverage)
```

Element	Description
<i>thinkAverage</i>	If specified as 0, the number of milliseconds stored in the <code>ThinkAvg</code> environment variable is entered. Otherwise, the value specified overrides <code>ThinkAvg</code> .

Return Value

On success, this function returns a calculated think-time average. A negative exit value indicates an error. Call `TSSErrorDetail()` for more information.

Comments

This call calculates and returns a think time using the same algorithm as `TSSThink()`. But unlike `TSSThink()`, 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 `TSSThinkTime()` to recalculate the delay requested by `TSSThink()` 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`IVValue iv;
iv.ivInt = TSSGetTime;
TSSInternalVarSet (IV_fcs_ts, iv);
TSSInternalVarSet (IV_lcs_ts, iv);
TSSInternalVarSet (IV_fcr_ts, iv);
TSSInternalVarSet (IV_lcr_ts, iv);
s32 pause = TSSThinkTime(5000);
```

See Also

`TSSThink()`

TSSThinkTime()

Test Script Console Adapter API

4

This chapter describes the Test Script Console Adapter (TSCA) and explains how to build a custom TSCA.

This chapter contains the following sections:

- About the Test Script Console Adapter
- Building a TSCA: Workflow and Implementation Issues
- Registering the TSCA DLL with TestManager

About the Test Script Console Adapter

A Test Script Console Adapter (TSCA) is a C or C++ dynamic-link library (DLL) that integrates with TestManager. By so doing, it enables additional test script types to be available for operations.

Each test script type is associated with a particular TSCA. A TSCA is required to allow the user to access a test script through the user interface. (Note that the TSCA does not support test script execution; this function is carried out by the TSEA and TSEE, as described in Chapter 2.)

TSCA Functionality

A TSCA must, at a minimum do the following:

- Connect to and disconnect from the test script source.
- Provide a way for the user to select a test script from the source.

In addition to these basic functions, many TSCAs are designed to support more sophisticated functions. A more robust TSCA might support such operations as the following:

- Displaying a directory hierarchy of test scripts.
- Filtering test scripts within the Test Script view.
- Displaying the properties of a test script.
- Configuring a test script source.

- Performing source control operations against the contents of a test script source.
- Executing a custom action against the test script source or the test script, for example, providing multiple editors for the same test script type.

Built-In and Custom Test Script Types

As described in Chapter 1, you can implement a test case with a test script that is either a built-in test script type or a custom test script type.

If you create a custom test script type, you must extend TestManager to support this new type. To extend TestManager, you either:

- Use the built-in Command Line TSCA or other console adapters provided by Rational.

The Command Line TSCA works for any file-based test script, for example, PERL scripts. File-based means that the individual test scripts can be accessed by their names or paths using the standard Microsoft File Open dialog box.

Because Rational provides this console adapter, you do not need to do any programming; you only need to specify the executable commands for creating and editing a test script.

Although this TSCA requires no custom programming, it is not fully integrated into TestManager. (For more information about using TestManager's built-in console adapter, see the TestManager online Help.)

- Create a custom test script console adapter.

A custom TSCA is required to integrate the custom test type with TestManager. Once you write the TSCA, TestManager will be able to recognize these new test script types.

A custom TSCA is required for test scripts that are:

- Created with a test tool that does not provide a command-line interface for creating and editing test scripts.
- Not file based, that is, cannot be opened with the standard File Open dialog box— for example, Rational ManualTest test scripts.

A custom TSCA can also be used with file-based test scripts.

Unlike the command-line TSCA, you can fully integrate a custom TSCA into TestManager.

The TSCA Function Calls

The TSCA applications programming interface (API) consists of 31 functions that are organized into nine functional groupings.

Functional Groupings of TSCA Functions

The Test Script Console Adapter (TSCA) functions are summarized in the following table, which shows

- The functional groupings and their purposes.
- The functions within each group.

Function Group and Purpose	Functions in group
Connection Supports connection and disconnection from the test script source.	TTConnect ()
	TTDisconnect ()
Data Access Provides access to the test scripts within the source: <ul style="list-style-type: none"> ▪ Hierarchical data ▪ Non-hierarchical data ▪ Sources with different test script types ▪ Functions providing icons for source and test scripts 	TTGetRoots ()
	TTGetChildren ()
	TTGetNode ()
	TTGetName ()
	TTGetTypeIcon ()
	TTGetSourceIcon ()
	TTGetIcon ()
Editor Integration Provides integration between TestManager and the editor or IDE used to create and edit test scripts.	TTNew ()
	TTEdit ()
Filtering Provides support to filter out test scripts that do not meet user-defined criteria.	TTSetFilterEx ()
	TTGetFilterEx ()
	TTClearFilter ()
UI Support Provides the ability to expose components that display test script properties. This facilitates the use of preexisting user interfaces.	TTShowProperties ()
	TTSelect ()

Function Group and Purpose	Functions in group
Source Configuration Support Supports the ability to provide custom user interfaces to aid in the configuration of the test script source.	TTGetConfiguration()
	TTSetConfiguration()
Custom Action Execution Provides the ability to perform custom operations on the test script source.	TTGetSourceActions()
	TTGetNodeActions()
	TTExecuteSourceAction()
	TTExecuteNodeAction()
Source Control Provides the ability to support source control operations in the test script view on the test scripts within a source.	TTAddToSourceControl()
	TTCheckIn()
	TTCheckOut()
	TTUndoCheckout()
	TTGetSourceControlStatus()
Execution Supports execution of the test script.	TTGetTestToolOptions()
Miscellaneous Infrastructure Provides infrastructure support	TTGetIsFunctionSupported()

Required and Optional Functionality

A TSCA must, at a minimum, provide the ability to:

- Connect to a test script source.
- Select a test script.

To enable these actions, you must implement the following functions when building a basic TSCA:

- TTConnect()
- TTDisconnect()
- TTSelect()

In addition, to enable users to view test scripts in Test Script View, you must include the following functions:

- `TTGetRoots()`
- `TTGetChildren()`

Other functions are optional; they enable the user to work with the user interface to perform additional operations on the test script. The advantages of using these additional functions are described in *Building a TSCA: Workflow and Implementation Issues* on page 116.

Some functions work in pairs. For example, because TestManager calls `TTConnect()` to make the connection to the test script source, it must subsequently call `TTDisconnect()` to disconnect from the test script source.

For information about specific declarations, see the following required header file:

```
...\Rational Test\rtsdk\c\include\testypeapi.h
```

Mapping of User Actions to TSCA Function Calls

A TestManager end user may want to carry out various actions on the test script source. Following is a list of common test script operations that the user might perform. The order in which these operations are listed represents a plausible sequence in which the user might execute them.

- 1 Defining or modifying the configuration of a test script source.
- 2 Opening the Test Script view.
- 3 Setting a filter for test scripts.
- 4 Creating a new test script.
- 5 Selecting a test script for operations.
- 6 Editing test script properties.
- 7 Editing test script text.
- 8 Performing custom actions on the test script or test script source.
- 9 Integrating with source control.

To aid the user in carrying out these actions, you should build the TSCA to support the test script view that enables the user to perform these operations.

When the user makes a selection in the GUI to carry out an operation on the test script source, TestManager typically calls the TSCA function or functions that support the user's action.

To help you understand which functions you need to implement in order to provide support for specific actions, the following sections explain the mapping between the user's actions and the TSCA functions called by TestManager to implement those actions.

Note: Based on the current state of TestManager (including which test-script-related operations it has already performed), TestManager may not call some of the functions listed in the following sequences.

Defining or Modifying the Configuration of a Test Script Source

When the user defines or modifies a configuration for a test script source (for example, specifying the operating system), TestManager calls the following functions from the **Source Configuration** group in the order listed.

Typical Sequence of Function Calls	Operation
1 TTGetConfiguration()	Exposes user interface elements that collect data access and data format information from the user when registering the test script source. The TSCA passes that information back to TestManager to be persisted as a property of the test script source.
2 TTSetConfiguration()	When a connection is made to the test script source, TestManager passes the configuration information obtained from the TTGetConfiguration() function into this function.

Opening the Test Script View

When the user opens up the Test Script view, TestManager calls the following functions from the **Connection** group in the order listed below.

Typical Sequence of Function Calls	Operation
1 TTConnect()	Establishes a connection to the test script source.
2 TTGetSourceIcon()	Returns the path to the bitmap containing the icon that represents the test script source. Note that implementation of this function is optional.

Setting a Filter for Test Scripts

When the user sets a filter on the test script source, TestManager calls the following functions, in the order shown below.

Creating a New Test Script

Typical Sequence of Function Calls	Operation
1 TTGetFilterEx()	Exposes user interface elements that collect filtering specifications from the user registering the test script source.
2 TTSetFilterEx()	When a connection is made to the test script source, TestManager passes the filtering specifications obtained from TTGetFilterEx() into the TSCA.
3 TTGetRoots()	Returns the array of nodes comprising the root elements of the test script source.

When the user creates a new test script, TestManager calls the following function from the **Editor Integration** group.

Function Call	Operation
TTNew()	Enables the tester to create a new test script for this type of test using the hosted tool.

Selecting a Test Script for Operations

Before carrying out any operations on a test script, the user selects the test script from a list of available test scripts in the Test Script view. To display these test scripts, TestManager calls the following functions from the **Data Access** group, typically in the sequence that follows.

Typical Sequence of Function Calls	Operation
1 TTGetSourceIcon()	Returns the path of the bitmap that represents the test script source.
2 TTGetRoots()	Returns the array of nodes comprising the roots of the test script source.
3 TTGetChildren()	Returns an array of nodes that are the children of the specified node. TestManager calls this function for each node returned by TTGetRoots().

Typical Sequence of Function Calls Operation

4	<code>TTGetTypeIcon()</code>	Returns the path of the bitmap that represents the test script node.
---	------------------------------	--

Editing Test Script Properties

When users want to edit the properties of a test script, they must:

- 1 Select the test script from the Test Script view.
- 2 Bring up the **Test Script Properties** dialog box to view and modify the test script properties.

To enable these actions, TestManager calls the following function, which comes from the **UI Support** group.

Function Call	Operation
<code>TTShowProperties()</code>	Displays the properties of a selected test script.

Editing Test Script Text

When the user selects operations to edit a test script, TestManager calls `TTEdit()`, which comes from the **Editor Integration** group.

Function Call	Operation
<code>TTEdit()</code>	Displays a test script in the appropriate editor for modification by the user.

Performing Custom Actions on the Test Script or the Test Script Source

You can implement the TSCA to support custom actions on the test script or on the test script source. If the TSCA supports these custom actions, TestManager displays them in a custom menu so that the user can select these actions.

To enable display and execution of custom actions on the test script source, TestManager calls these functions from the **Custom Action Execution** group, typically in the following sequence.

Typical Sequence of Function Calls	Operation
1 TTGetSourceActions ()	Returns a pointer to an array of actions that can be applied to the test script source.
2 TTExecuteSourceAction ()	Executes the specified action against the test script source.

To enable display and execution of custom actions on a test script, TestManager calls these functions from the **Custom Action Execution** group, typically in the following sequence.

Typical Sequence of Function Calls	Operation
1 TTGetNodeActions ()	Returns a pointer to an array of actions that can be applied to the test script.
2 TTExecuteNodeAction ()	Executes the specified action against the test script.

Integrating with Source Control

When the user makes selections in the UI to integrate a test script with source control, TestManager calls the following functions from the TSCA. These functions come from the **Source Control** group. TestManager typically calls TTGetSourceControlStatus () first and then calls the other functions in an order that corresponds to the order of the user's selections in the UI.

Typical Sequence of Function Calls	Operation
1 TTGetSourceControlStatus ()	Returns the current source-control status of the test script.
2 One of the following:	
▪ TTAddToSourceControl ()	Adds the appropriate files for the specified test script to source control.
▪ TTCheckout ()	Checks out the appropriate files for the specified test script from source control.
▪ TTCheckIn ()	Checks in the appropriate files for the specified test script to source control.
▪ TTUndoCheckout ()	Undoes the checkout of the appropriate files for the specified test script.

Building a Custom Test Script Console Adapter

This section describes

- Skills you need to build a custom TSCA.
- Implementation issues that arise when you create this adapter.

Prerequisite Skills

To build a custom TSCA using TestManager's C/C++ API, you need the following skills

- A working knowledge of the tool used to create test scripts of the given type. Especially important is a knowledge of how to programmatically access the test scripts within the test script source.
- An ability to build a C or C++ DLL that exposes the functions called by TestManager.
- Familiarity with the TSCA API.

Building a TSCA: Workflow and Implementation Issues

This section discusses the general workflow that you should follow when building a TSCA and the implementation issues that arise at each phase of this workflow. The phases are

- Making a connection.
- Accessing the data.
- Integrating with source control.
- Displaying properties.
- Filtering.
- Custom action support.

Making a Connection

The most critical phase in developing a TSCA is determining how to use the functions in the Data Access group to access the data (that is, the test scripts stored in a source) and return this data to TestManager. In most cases, the data is accessed using a common API or directly by accessing the physical representation of the data. This physical representation can be a file, a database, or some other source.

To work efficiently, the TSCA should maintain an active connection to the test script source rather than reconnecting each time the user makes a request from TestManager.

Given these considerations, you, the TSCA developer, must determine the answers to the following connection issues at the beginning, before you create the TSCA. The decisions you make regarding the connection issues are the most critical decisions you make in the process of building the TSCA. Once you have determined satisfactory solutions to the following issues, you are likely to be successful in your development of the TSCA.

- 1 How can the TSCA gain access to the test scripts in the source? If an existing API performs this function, you should probably use it. If an appropriate API does not exist, is there a way to gain access to the data directly?
- 2 How can the adapter uniquely identify the test script source? If the test scripts are stored in the file system, the preferred form of identification is the root path to the files. The test script source identification that you develop is passed to the `TTConnect()` function when TestManager calls it.
- 3 Is any additional information needed for connecting to the source? If so, you should specify that information as a connection option when registering the test script source. The connection options are passed to the `TTGetTestToolOptions()` function when TestManager calls it.
- 4 How can the TSCA uniquely identify a test script? If the tool used to create the test script has a feature that is analogous to a unique ID, it is optimal to use this feature. Note that each test script must have its own unique ID for each test script source, because that identifier is used to associate test cases with it.
- 5 Does the test script source require any configuration data in addition to the data needed for connection? If so, what data is needed? In developing solutions to these issues, you should implement the functions `TTSetConfiguration()` and `TTGetConfiguration()`.

Accessing the Data

The next phase in building a TSCA is to develop support for the needed functions in the Data Access group.

Once this support is developed, the user should be able to do the following after a new test script type has been registered to use the TSCA

- Register the test script source.
- Use Test Script view to view the test scripts in that source.

The issues to resolve in developing support for needed functions in the Data Access group are as follows:

- 1 What is the organizational structure for the test scripts that the TSCA will access? Is the data organized in a hierarchical structure or in a flat list?

If the test scripts are organized in a flat list, you need the following functions to return data:

- `TTGetRoots()`
- `TTGetNode()`

If the test scripts are stored hierarchically, you need the following functions to return data:

- `TTGetRoots()`
- `TTGetNode()`
- `TTGetChildren()`

- 2 Are the items contained in the test script source all of the same type? Are they all valid implementations that can be executed? (Examples of items that cannot be implemented are verification points and low-level test scripts contained in Rational Robot GUI test scripts.) Even if certain items cannot be executed, you may want to make them visible to TestManager by implementing the following functions:

- `TTGetRoots()`
- `TTGetChildren()`

- 3 Is there an easily recognized bitmap that represents the test script source?

Because Windows users are accustomed to recognizing software components by icons, you should support the function `TTGetSourceIcon()`. This function returns the path of the bitmap that represents the test script source.

- 4 Are there easily recognized bitmaps that represent the different elements comprising test scripts in the test script source?

Because Windows users are accustomed to recognizing software components by icons, you should support the function `TTGetTypeIcon()`. This function returns the path of the bitmap that represents the test script node.

Integration with Source Control

To support integration with source control, TestManager needs the TSCA to support the following functions:

- `TTCheckIn ()`
- `TTCheckOut ()`
- `TTAddToSourceControl ()`
- `TTUndoCheckout ()`
- `TTGetSourceControlStatus ()`

You must decide which files need to be placed under source control.

Displaying Properties

By default, the only test script information displayed by TestManager's user interface is the test script's name. If you want to enable the user to view other associated data about the test script, you should implement the function `TTShowProperties ()`. If possible, use the underlying tool's property sheet if it is appropriate for display and can be called by the TSCA.

Supporting User Configuration of the Test Script Source

The formats of different test script sources vary as to how the adapter accesses the information in them. For example, if test scripts are stored in a database, the test scripts may be stored in different tables for different test script sources.

Therefore, you may want the TSCA to provide a GUI that enables the user to provide configuration information when registering the test script source. By specifying configuration data, the user is informing TestManager about how to locate and identify the desired test scripts in the test script source.

If you decide that you need to provide this configuration ability to the user, you must decide:

- What kind of configurability the user needs.
- What kind of user interface to provide to the user for specifying the configuration.

Thus, when the test script sources are highly variable, write the TSCA to display a GUI that enables the user to specify the test script storage configuration and communicate it to TestManager. To enable this functionality, implement the following functions:

- `TTGetConfiguration()`

This function prompts the user with a user interface that collects the information needed to configure the adapter. This function must return that information in a buffer so that TestManager can persist it with the test script source.

- `TTSetConfiguration()`

This function enables TestManager to pass the test script source configuration information that is collected by `TTGetConfiguration` into the TSCA. The TSCA needs this information to know how to access the data in the test script source.

In summary, consider the type of test script source when deciding whether to provide configurability to the user.

Filtering

If a test script source contains an enormous number of test scripts, you should provide filtering support. Filtering enables the user to use the Test Script view to selectively view the test scripts in the test script source. If the test script is created in a tool that supports filtering, take advantage of it.

If you implement the following functions, TestManager can use filtering in the Test Script view:

- `TTSetFilterEx()`

- `TTGetFilterEx()`

Custom Action Support

TestManager enables a TSCA to expose operations in the Test Script view that the test designer can use when building test scripts. For example, a test designer working with a Rational Robot test script as a test script source would probably want to carry out an operation that opens the Robot application.

You, the adapter writer, can build the adapter so that the user can implement an operation to open Robot from the GUI. You can also support custom operations for specific test scripts.

To support custom actions against a test script source, implement the following functions:

- `TTGetSourceActions()`
- `TTExecuteSourceAction()`

To support custom actions against a test script, implement the following functions:

- `TTGetNodeActions()`
- `TTExecuteNodeAction()`

Registering the TSCA DLL with TestManager

If you have created your own TSCA, you must let TestManager know it exists. To register a DLL:

- 1 In TestManager click **Tools > Manage > Test Script Types**. The Console Adapter Type dialog box appears.
- 2 Click **Use a custom console adapter**.
- 3 Type the path to the DLL in the space provided, labeled **Console adapter DLL file**.

For more information, see the TestManager online Help.

TSCA Function Reference

This section of the Test Script Console Adapter chapter provides reference information for all functions in the TSCA API. For each function, the following information is presented:

- Definition
- Syntax
- Return values
- Comments (if any)
- Example
- See Also (links to related functions)

Note: The following functions are not currently used but are defined in the header file and reserved for future use.

TTAddToSourceControl()

```
TTCompile()  
TTGetIsChild()  
TTGetIsParent()  
TTGetIsValidSource()  
TTGetName()  
TTGetParent()  
TTGetType()  
TTGetTypes()  
TTRecord()
```

TTAddToSourceControl()

Adds the appropriate files for the specified test script to source control.

Syntax

```
HRESULT TTAddToSourceControl(const TCHAR  
    SourceID[TTYPE_MAX_PATH], const TCHAR NodeID[TTYPE_MAX_ID],  
    long lWindowContext, TCHAR  
    ErrorDescription[TTYPE_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>NodeID</i>	INPUT. The unique ID that identifies the selected test script in the source.
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this method.
<i>ErrorDescription</i>	OUTPUT. An error description that is returned by the adapter.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_CANCEL. The user pressed the Cancel button.
- TTYPE_ERROR_INVALID_SOURCEID. The test script source was incorrectly identified.

- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Example

```

/*****
TESTTYPEAPI API HRESULT TTAddToSourceControl(const TCHAR
SourceID[TTYPE_MAX_ID], const TCHAR NodeID[TTYPE_MAX_ID], long
lWindowContext, TCHAR ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TTYPE_ERROR_INVALID_SOURCEID;

    try
    {
        /* CODE OMITTED: Retrieve assets associated with NodeID.
        Perform add to source control on
        assets. */

    }
    catch (_com_error &e)
    {
        rc = TTYPE_ERROR;
        CString sError = (BSTR) e.Description();
        tcscopy(ErrorDescription, (LPCTSTR) sError );
    }
    return eResult;
}

```

See Also

TTCheckIn(), TTCheckOut(), TTGetSourceControlStatus(),
TTUndoCheckout()

TTCheckIn()

Checks in the appropriate files for the specified test script to source control.

Syntax

```
HRESULT TTCheckIn(const TCHAR SourceID[TTYPE_MAX_PATH], const
    TCHAR NodeID, long lWindowContext, TCHAR
    ErrorDescription[TTYPE_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>NodeID</i>	INPUT. The unique ID that identifies the selected test script in the source.
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this method.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_CANCEL. The user pressed the Cancel button.
- TTYPE_ERROR_INVALID_SOURCEID. The test script source was incorrectly identified.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Example

```
//*****
TESTTYPEAPI_API HRESULT TTCheckIn(const TCHAR SourceID[TTYPE_MAX_ID],
const TCHAR NodeID[TTYPE_MAX_ID], long lWindowContext, TCHAR
ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;
    CConnectionContext *pContext=0;
```

```

// lookup the context information for the specified SourceID
CString sSourceID = SourceID;
m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

if (pContext == 0)
    return TTYPE_ERROR_INVALID_SOURCEID;

try
{
    /* CODE OMITTED: Retrieve assets associated with NodeID.
    Perform source control CheckIn on
    assets. */

}
catch (_com_error &e)
{
    rc = TTYPE_ERROR;
    CString sError = (BSTR) e.Description();
    tcscopy(ErrorDescription, (LPCTSTR) sError );
}
return rc;
}

```

See Also

TTAddToSourceControl(), TTCheckout(),
TTGetSourceControlStatus(), TTUndoCheckout()

TTCheckout()

Checks out, from source control, the appropriate files for the specified test script.

Syntax

```

HRESULT TTCheckout(const TCHAR SourceID[TTYPE_MAX_PATH], const
    TCHAR NodeID, long lWindowContext, TCHAR
    ErrorDescription[TTYPE_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>NodeID</i>	INPUT. The unique ID that identifies the selected test script in the source.

Element	Description
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this method.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_CANCEL. The user pressed the Cancel button.
- TTYPE_ERROR_INVALID_SOURCEID. The test script source was incorrectly identified.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Example

```
//*****
TESTTYPEAPI_API HRESULT TTCheckout(const TCHAR SourceID[TTYPE_MAX_ID],
const TCHAR NodeID[TTYPE_MAX_ID], long lWindowContext, TCHAR
ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TTYPE_ERROR_INVALID_SOURCEID;

    try
    {
        /* CODE OMITTED: Retrieve assets associated with NodeID.
        Perform source control CheckOut on assets. */
    }
    catch (_com_error &e)
    {
        rc = TTYPE_ERROR;
    }
}
```

```

        CString sError = (BSTR) e.Description();
        tcscopy(ErrorDescription, (LPCTSTR) sError );
    }
    return rc;
}

```

See Also

TTAddToSourceControl(), TTCheckIn(),
TTGetSourceControlStatus(), TTUndoCheckout()

TTClearFilter()

Clears the filter for the test script source.

Syntax

```

HRESULT TTClearFilter(const TCHAR SourceID[TTYPE_MAX_ID], TCHAR
    ErrorDescription[TTYPE_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The handle that the client uses to identify the connection to the datastore.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Example

```

//*****
TESTTYPEAPI_API HRESULT TTClearFilter(const TCHAR
SourceID[TTYPE_MAX_ID], TCHAR ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;

```

TTConnect()

```
CConnectionContext *pContext=0;

// lookup the context information for the specified SourceID
CString sSourceID = SourceID;
m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

if (pContext == 0)
    return TTYPE_ERROR_INVALID_SOURCEID;

/* The following lines are for a file-based filter mechanism.
The filter buffer is first cleared, followed by replacing the
derived list of supported file extensions with *.* */

// Clear the filter buffer
pContext->m_sFilter = "";

pContext->m_asFilterFileExtensions.RemoveAll();
pContext->m_asFilterFileExtensions.Add("*.");

return rc;
}
```

See Also

TTSetFilterEx(), TTGetFilterEx()

TTConnect()

Creates a connection to the script source identifies by the contents of the *ConnectInfo* paramater. It returns a handle to the client for subsequent calls to the adapter.

Syntax

```
HRESULT TTConnect(const char ConnectInfo[TTYPE_MAX_PATH], const
char UserID[TTYPE_MAX_ID], ConnectOption *pConnectOptions[],
int nOptions, char SourceID[TTYPE_MAX_ID], char
ErrorDescription[TTYPE_MAX_ERROR])
```

Element	Description
<i>ConnectInfo</i>	INPUT. Contains the data path to the test script source.
<i>UserID</i>	INPUT. The ID of the user to connect to the datastore.
<i>pConnectOptions</i>	INPUT. A pointer to the defined connect options.
<i>nOptions</i>	INPUT. An integer that specifies the total number of connection options.

Element	Description
<i>SourceID</i>	OUTPUT. The handle that TestManager uses to identify the connection to the datastore in subsequent calls to the adapter.
<i>ErrorDescription</i>	OUTPUT. An message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TTYTYPE_SUCCESS`. The function completed successfully.
- `TTYTYPE_ERROR_UNABLE_TO_CONNECT`. The connection failed for some unknown reason.
- `TTYTYPE_ERROR_INVALID_CONNECTINFO`. The adapter was unable to use the connection information.
- `TTYTYPE_ERROR`. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

The data type `ConnectOption` is defined as follows:

```
struct ConnectOption
{
    char Name[TTYTYPE_MAX_NAME];
    char Value[TTYTYPE_MAX_NAME];
}ConnectOptionType;
```

The connection information is specified by an administrative user in the TestManager New Test Script Source property page and then passed into this function in *ConnectInfo*. After the connection has been established, the TSCA assigns a unique identifier for the test source to *SourceID*. TestManager uses this identifier for subsequent calls to the adapter to identify the connection. Be sure to document the format of this string.

Example

```
//*****
TESTTYPEAPI_API HRESULT TTConnect (const char
ConnectInfo[TTYTYPE_MAX_PATH], char UserID[TTYTYPE_MAX_ID], ConnectOption
*pConnectOptions, int nOptions, char SourceID[TTYTYPE_MAX_ID], char
ErrorDescription[TTYTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE (AfxGetStaticModuleState ());
```

TTConnect()

```
HRESULT rc = TTYPE_SUCCESS;

if (_tcslen(ConnectInfo) == 0)
{
    CString sError;
    sError.LoadString(IDS_EMPTY_PATH);
    memset(ErrorDescription, _T('\0'), sizeof (ErrorDescription) );
    _tcsncpy(ErrorDescription, sError, TTYPE_MAX_ERROR - 1);
    rc = TTYPE_ERROR;
}
else
{
    CFileFind finder;
    BOOL bExists = finder.FindFile(ConnectInfo);
    if (!bExists)
    {
        CString sError;
        AfxFormatString1(sError, IDS_ERROR_NOSOURCE, ConnectInfo);
        memset(ErrorDescription, _T('\0'), sizeof (ErrorDescription));
        _tcsncpy(ErrorDescription, sError, TTYPE_MAX_ERROR - 1);
        rc = TTYPE_ERROR;
        return rc;
    }

    // Need to go through pConnectionOption the identifier names to
    // for the edit and new
    CString sNewCommand = pConnectOptions[0].Value;
    CString sEditCommand = pConnectOptions[1].Value;
    CString sUID = pConnectOptions[2].Value;

    // Generate a unique connection id for this connection context
    CString sConnectionIdentifier;
    GUID newGuid;
    CoCreateGuid(&newGuid); // note: this method is Windows-only
    sConnectionIdentifier.Format("%x-%x-%x-%x%x%x%x%x%x%x",
        newGuid.Data1, newGuid.Data2, newGuid.Data3,
        newGuid.Data4[0], newGuid.Data4[1], newGuid.Data4[2],
        newGuid.Data4[3],
        newGuid.Data4[4], newGuid.Data4[5], newGuid.Data4[6],
        newGuid.Data4[7] );

    CConnectionContext *pExistingContext = 0;

    // Not connected yet.
    try
    {
        CString sUserName, sPassword;

        //For this example, assume that connection options include
        //the UserName and Password necessary for connecting to the
        //datastore.
    }
}
```



```

// Retrieve the UserID and Password.
for (int iIndex = 0; iIndex < nOptions; iIndex++)
{
    if (_tcsicmp(pConnectOptions[iIndex].Name, "UserName")
        == 0)
    {
        sUserName = pConnectOptions[iIndex].Value;
    }
    else
    {
        if (_tcsicmp(pConnectOptions[iIndex].Name, "Password")
            == 0)
        {
            sPassword = pConnectOptions[iIndex].Value;
        }
    }
}

/* CODE OMITTED: Attempt to establish a connection to test
script data store using the UserName and Password connection
options. In some cases, connecting to a data store may not
require any connection options.*/

// Store the connection context in the connection map.
m_ServerConnections.SetAt(sConnectionIdentifier, pContext);

} // for loop
catch (_com_error)
{
    rc = TTYPE_ERROR_UNABLE_TO_CONNECT;
    _tcscopy(ErrorDescription, (LPCTSTR) "Connection failure -
Bad username and Password");
}
} // else

return rc;
}

```

See Also

TTDisconnect()

TTDisconnect()

TTDisconnect()

Disconnects from an existing script source.

Syntax

```
HRESULT TTDisconnect(char SourceID[TTYTYPE_MAX_ID], char  
                  ErrorDescription[TTYTYPE_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle that the client uses to identify the connection to the datastore.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TTYTYPE_SUCCESS`. The function completed successfully.
- `TTYTYPE_ERROR_INVALID_SOURCEID`. The specified source information was not correct.
- `TTYTYPE_ERROR_UNABLE_TO_DISCONNECT`. There was no existing connection to disconnect from.
- `TTYTYPE_ERROR`. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

After TestManager calls `TTDisconnect()`, no further operations are allowed on this test script source.

Example

```
//*****  
HRESULT TTDisconnect(const CHAR SourceID[TTYTYPE_MAX_ID], CHAR  
ErrorDescription[TTYTYPE_MAX_ERROR])  
{  
    AFX_MANAGE_STATE(AfxGetStaticModuleState());  
  
    HRESULT rc = TTYTYPE_SUCCESS;  
  
    CConnectionContext *pContext=0;
```

```

// Lookup the context information for the specified SourceID.
CString sSourceID = SourceID;
m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

// If the context was found, then continue else indicate SourceID is
// invalid.
if (pContext)
{
/* CODE OMITTED: Disconnect from test script datastore.
   If unable to disconnect, return
   TTYPE_ERROR_UNABLE_TO_DISCONNECT.*/
}
else
    rc = TTYPE_ERROR_INVALID_SOURCEID;

return rc;
}

```

See Also

TTConnect ()

TTEdit()

Launches the editor for a specific test script.

Syntax

```

HRESULT TTEdit(const char SourceID[TTYPE_MAX_ID], const char
    ScriptID[TTYPE_MAX_ID], int LineNumber, ScriptOption
    *pScriptOptions[], int *nScriptOptions, char
    ErrorDescription[TTYPE_MAX_ERROR], long lWindowContext)

```

Element	Description
<i>SourceID</i>	INPUT. A string that identifies the connection.
<i>ScriptID</i>	INPUT. A string that identifies the test script.
<i>LineNumber</i>	INPUT. An integer that specifies where the cursor is placed when the file is opened for editing.
<i>pScriptOptions</i>	INPUT. The test script options passed from TTSelect (). Any changes made to the test script options are passed back and made available to the test case or suite.
<i>nScriptOptions</i>	INPUT. An integer that specifies the number of test script options.

TTEdit()

Element	Description
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.
<i>lWindowContext</i>	INPUT. An integer that specifies the handle for the parent window (HWND)

Return Values

This function typically returns one of the following values:

- `TTYTYPE_SUCCESS`. The function completed successfully.
- `TTYTYPE_ERROR_INVALID_SOURCEID`. The test source was incorrectly identified.
- `TTYTYPE_ERROR_INVALID_ID`. The adapter could not find a test script with this identification.
- `TTYTYPE_ERROR`. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

Script options are settings that can sometimes be important for the test script to execute properly. The TSCA can specify and modify script options that TestManager saves as part of the test case. The data type `ScriptOption` is defined as follows:

```
struct ScriptOption
{
    char Name[TTYTYPE_MAX_NAME];
    char Value[TTYTYPE_MAX_NAME];
}ScriptOptionType;
```

Example

```
//*****
HRESULT TTEdit(const CHAR SourceID[TTYTYPE_MAX_ID],const CHAR
ScriptID[TTYTYPE_MAX_ID], ScriptOption *pScriptOptions[], int
*nScriptOptions, CHAR ErrorDescription[TTYTYPE_MAX_ERROR], long
lWindowContext)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYTYPE_SUCCESS;

    CConnectionContext *pContext=0;

    // Lookup the context information for the specified SourceID.
    CString sSourceID = SourceID;
```

```

m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

// If the context was found, continue.
if (pContext)
{
    /* CODE OMITTED: Open the test script identified by the value of
    parameter ScriptID. This could be as simple as executing a
    command line call to your favorite text editor or a more complex
    interaction with a test tool.*/

}
else
    rc = TTYPE_ERROR_INVALID_SOURCEID;

return rc;
}

```

See Also

TTNew(), TTShowProperties(), TTSelect()

TTEecuteNodeAction()

Executes the specified action against the specified test script.

Syntax

```

HRESULT TTEecuteNodeAction(const TCHAR SourceID[TT_MAX_PATH],
    const TCHAR Node[TT_MAX_ID], int nActionID, long
    lWindowContext, TCHAR ErrorDescription[TT_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test script source.
<i>Node</i>	OUTPUT. The ID that identifies the selected test script in the test script source.
<i>nActionID</i>	INPUT. The ID of the action to be executed.
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this function.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

TTEecuteSourceAction()

Return Values

This function typically returns one of the following values:

- TT_SUCCESS. The function completed successfully.
- TT_ERROR. The function did not complete successfully.

See Also

TTGetSourceActions(), TTGetNodeActions(),
TTEecuteSourceAction()

TTEecuteSourceAction()

Executes the specified action against the test script source.

Syntax

```
HRESULT TTEecuteSourceAction(const TCHAR  
    SourceID[TTYE_MAX_PATH], int nActionID, long  
    lWindowContext, TCHAR ErrorDescription[TTYE_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>nActionID</i>	INPUT. The ID of the action to be executed.
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this method.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYE_SUCCESS. The function completed successfully.
- TTYE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

The adapter executes the specified action against the test script source.

The following structure supports this function:

```
#define TTYPE_ACTION_NAME      100

struct Action
{
    char    Name[TTYPE_ACTION_NAME]:
    int     Action ID;
} ActionType;
```

Example

```
//*****
TESTTYPEAPI_API HRESULT TTEecuteSourceAction(const TCHAR
SourceID[TTYPE_MAX_ID], int nActionID, long lWindowContext, TCHAR
ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TTYPE_ERROR_INVALID_SOURCEID;

    switch (nActionID)
    {
        case 1:
            /* CODE OMITTED: Execute custom action 1. */
            break;
        case 2:
            /* CODE OMITTED: Execute custom action 2. */
            break;

        default:
            rc = TTYPE_ERROR;
            _tcscpy(ErrorDescription, _T("Unrecognized action
            received"));
            break;
    }
    return rc;
}
```

TTGetChildren()

See Also

TTGetNodeActions(), TTEecuteNodeAction(),
TTEecuteSourceAction()

TTGetChildren()

Returns an array of nodes that are the children of the specified node.

Syntax

```
HRESULT TTGetChildren(const TCHAR SourceID[TTYPE_MAX_PATH],  
    const TCHAR NodeID[TTYPE_MAX_ID], struct Node  
    *pChildNodes[], long *plNodeCount, TCHAR  
    ErrorDescription[TTYPE_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>NodeID</i>	INPUT. The unique ID that identifies the selected test script in the source.
<i>plChildNodes</i>	OUTPUT. An array of populated Node structures, each of which is a root of the source.
<i>plNodeCount</i>	OUTPUT. The number of nodes returned by this function.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

The following structure supports this function.

```
struct ScriptNode
{
    TCHAR    Name [TTYPE_MAX_NAME];
    TCHAR    NodeID [TTYPE_MAX_ID];
    TCHAR    Type [TTYPE_MAX_TYPE];
    BOOL     IsOnlyContainer;
    BOOL     IsImplementation;
};
```

Example

```
//*****
TESTTYPEAPI_API HRESULT TTGetChildren(const TCHAR
SourceID [TTYPE_MAX_ID], const TCHAR NodeID [TTYPE_MAX_ID], struct
ScriptNode *pChildNodes [], long* plNodeCount, TCHAR
ErrorDescription [TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE (AfxGetStaticModuleState ());

    HRESULT rc = TTYPE_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup (sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TTYPE_ERROR_INVALID_SOURCEID;

    // In this example, insert a single child node
    *pNodeArray = new struct ScriptNode [1];
    *plNodeCount = 1;
    *pChildNodes = 0;

    (*pNodeArray) [0].IsImplementation = TRUE;
    (*pNodeArray) [0].IsOnlyContainer = FALSE;
    _tcscpy ((*pNodeArray) [0].Type, _T ("Node"));
    _tcscpy ((*pNodeArray) [0].Name, sName);
    _tcscpy ((*pNodeArray) [0].NodeID, _T ("C:\\RootFolder\\Node"));

    return rc;
}
```

TTGetConfiguration()

See Also

TTGetNode(), TTGetRoots(), TTGetSourceIcon(), TTGetTypeIcon()

TTGetConfiguration()

Returns a pointer to a buffer that contains a persistable configuration for the test script source.

Syntax

```
HRESULT TTGetConfiguration(const TCHAR  
    SourceID[TTYPE_MAX_PATH], long lWindowContext, TCHAR  
    **pConfigurationBuffer, int *pnConfigurationBufferLength,  
    TCHAR ErrorDescription[TTYPE_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this method.
<i>pConfigurationBuffer</i>	OUTPUT. A pointer to the buffer that contains the streamed configuration.
<i>pnConfigurationBufferLength</i>	OUTPUT. The length of the configuration buffer.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TTYPE_SUCCESS`. The function completed successfully.
- `TTYPE_ERROR`. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

The adapter returns the buffer configuration as a stream of characters. This data is interpreted by the adapter only.

It is assumed that the configuration buffer is allocated by the adapter and deleted by TestManager.

The adapter can display a user interface if necessary.

TestManager is responsible for persisting the data.

Example

```
//*****
TESTTYPEAPI_API HRESULT TTGetConfiguration(const TCHAR
SourceID[TTYPE_MAX_ID], long lWindowContext, TCHAR
**pConfigurationBuffer, int* pnConfigurationBufferLength, TCHAR
ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified
    // SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TTYPE_ERROR_INVALID_SOURCEID;

    *pFilterBuffer = new char[_MAX_PATH];

    CWnd ParentWnd;
    ParentWnd.FromHandle((HWND)lWindowContext);

    /* You have to create a class (A dialog with an embedded
    list control to perform configuration selection is used here)*/
    CSelectConfigDialog Dialog(pContext, &ParentWnd);

    if (Dialog.DoModal() == IDOK)
    {
        _tcscpy(*pConfigurationBuffer,
        Dialog.m_sSelectedConfigurationName);
        *pnConfigurationBufferLength =
        Dialog.m_sSelectedConfigurationName.GetLength();
    }
    else
    {
        // Put in blanks to indicate no filter set
    }
}
```

TTGetFilterEx()

```
        CString sEmpty;
        _tcscpy(*pConfigurationBuffer, sEmpty);
        *pnConfigurationBufferLength = sEmpty.GetLength();
    }

    return rc;
}
```

See Also

TTSetConfiguration()

TTGetFilterEx()

Returns a buffer containing a filter for the test script source.

Syntax

```
HRESULT TTGetFilterEx(const TCHAR SourceID[TTYPE_MAX_PATH],
    long lWindowContext, TCHAR *pFilterBuffer, int
    *pnFilterBufferLength, TCHAR
    ErrorDescription[TTYPE_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this method.
<i>pFilterBuffer</i>	OUTPUT. A pointer to the buffer that contains the streamed filter.
<i>pnFilterBufferLength</i>	OUTPUT. The length of the filter buffer.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

The filter contained in the buffer is returned as a stream of characters.

The data is interpreted only by the adapter.

TestManager is responsible for persisting the data.

The adapter can display a user interface if desired.

Example

```

//*****
TESTTYPEAPI_API HRESULT TTGetFilterEx(const TCHAR
SourceID[TTYE_MAX_ID], long lWindowContext, TCHAR **pFilterBuffer,
int* pnFilterBufferLength, TCHAR ErrorDescription[TTYE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TTYPE_ERROR_INVALID_SOURCEID;

    *pFilterBuffer = new char[_MAX_PATH];

    CWnd ParentWnd;
    ParentWnd.FromHandle((HWND)lWindowContext);

    /* You have to create a class (Typically a dialog with an
    embedded list control to perform filter selection here) */
    CSelectQueryDialog Dialog(pContext, &ParentWnd);

    if (Dialog.DoModal() == IDOK)
    {
        _tcscpy(*pFilterBuffer, Dialog.m_sSelectedQueryName);
        *pnFilterBufferLength =
            Dialog.m_sSelectedQueryName.GetLength();
    }
    else
    {
        // Put in blanks to indicate no filter set
        CString sEmpty;
        _tcscpy(*pFilterBuffer, sEmpty);
        *pnFilterBufferLength = sEmpty.GetLength();
    }
}

```

```

TTGetIcon()

    return rc;
}

```

See Also

TTSetFilterEx()

TTGetIcon()

Returns the path of the bitmap that represents test scripts in the source.

Syntax

```

HRESULT TTGetIcon(const char SourceID[TTYPE_MAX_ID], char
    IconPath[TTYPE_MAX_PATH], char
    ErrorDescription[TTYPE_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. A string that identifies the connection.
<i>IconPath</i>	OUTPUT. A string that specifies the path where the image file of the icon is located. The file must contain a 16 x 16 bitmap image.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_ERROR_INVALID_SOURCEID. The test script source was incorrectly identified.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Example

```

//*****
TESTTYPEAPI_API HRESULT TTGetIcon(char SourceID[TTYPE_MAX_ID], char
IconPath[TTYPE_MAX_PATH], char ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
}

```

```

HRESULT rc = TTYPE_SUCCESS;

DWORD dwError;
CString sIcon;
char szModuleFileName[_MAX_PATH+1];
char szModuleFilePath[_MAX_PATH+1];
char szDir[_MAX_DIR+1];
char szDrive[_MAX_DRIVE+1];

// In this sample, the icon is assumed to be in the same location as
// the adapter
dwError=GetModuleFileName((HMODULE)AfxGetInstanceHandle(),
szModuleFileName, _MAX_PATH);
_splitpath(szModuleFileName, szDrive, szDir, NULL, NULL);

_tcscpy(szModuleFilePath, szDrive);
_tcscat(szModuleFilePath, szDir);

sIcon.LoadString(IDS_ICON_NAME);
_tcscat(szModuleFilePath, sIcon);
_tcscpy(IconPath, szModuleFilePath);

return rc;
}

```

See Also

TTEdit(), TTNew()

TTGetIsFunctionSupported()

Indicates whether a specified function is supported by the adapter for an active connection.

Syntax

```

HRESULT TTGetIsFunctionSupported(const TCHAR
    SourceID[TTYPE_MAX_PATH], long lFunctionID, BOOL
    *pbSupported, TCHAR ErrorDescription[TTYPE_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>lFunctionID</i>	INPUT. The constant definition of the function.

Element	Description
<i>pbSupported,</i>	OUTPUT. A boolean indicating whether the specified function is supported.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TTYPE_SUCCESS`. The adapter supports *lFunctionID*.
- `TTYPE_ERROR`. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

Following are the values for the parameter *lFunctionID*:

```

TTYPE_FUNCTION_TTAddToSourceControl
TTYPE_FUNCTION_TTCheckIn
TTYPE_FUNCTION_TTCheckOut
TTYPE_FUNCTION_TTCompile
TTYPE_FUNCTION_TTConnect
TTYPE_FUNCTION_TTDisconnect
TTYPE_FUNCTION_TTEdit
TTYPE_FUNCTION_TTExecuteNodeAction
TTYPE_FUNCTION_TTExecutionSourceAction
TTYPE_FUNCTION_TTGetChildren
TTYPE_FUNCTION_TTGetConfiguration
TTYPE_FUNCTION_TTGetFilterEx
TTYPE_FUNCTION_TTGetIcon
TTYPE_FUNCTION_TTGetName
TTYPE_FUNCTION_TTGetNode
TTYPE_FUNCTION_TTGetNodeActions
TTYPE_FUNCTION_TTGetRoots
TTYPE_FUNCTION_TTGetSourceActions
TTYPE_FUNCTION_TTGetSourceControlStatus
TTYPE_FUNCTION_TTGetSourceIcon
TTYPE_FUNCTION_TTGetTestToolOptions
TTYPE_FUNCTION_TTGetTypeIcon
TTYPE_FUNCTION_TTNew
TTYPE_FUNCTION_TTRecord
TTYPE_FUNCTION_TTSelect
TTYPE_FUNCTION_TTSetConfiguration
TTYPE_FUNCTION_TTSetFilterEx
TTYPE_FUNCTION_TTShowProperties
TTYPE_FUNCTION_TTUndoCheckOut

```


Example

```

//*****
TESTTYPEAPI_API HRESULT TTGetIsFunctionSupported(const TCHAR
SourceID[TTYPE_MAX_ID], const TCHAR NodeID[TTYPE_MAX_ID], long
lFunctionID, BOOL *pbIsSupported, TCHAR
ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    /* The following is a sample for a minimal adapter with no source
    control support. */

    switch (lFunctionID)
    {
        case TTYPE_FUNCTION_TTConnect:
        case TTYPE_FUNCTION_TTDisconnect:
        case TTYPE_FUNCTION_TTEdit:
        case TTYPE_FUNCTION_TTGetIcon:
        case TTYPE_FUNCTION_TTGetName:
        case TTYPE_FUNCTION_TTNew:
        case TTYPE_FUNCTION_TTRecord:
        case TTYPE_FUNCTION_TTSelect:
        case TTYPE_FUNCTION_TTShowProperties:
            *pbIsSupported = TRUE;
            break;

        default:
            *pbIsSupported = FALSE;
            break;
    }

    return TTYPE_SUCCESS;
}

```

TTGetName()

Returns the name of the specified test script.

Syntax

```

HRESULT TTGetName(const char SourceID[TTYPE_MAX_ID], const char
ScriptID[TTYPE_MAX_ID], char ScriptName[TTYPE_MAX_ID], char
ErrorDescription[TTYPE_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. A string that identifies the connection.

TTGetName()

Element	Description
<i>ScriptID</i>	INPUT. A string that identifies the test script.
<i>ScriptName</i>	OUTPUT. A string that specifies the name of a test script.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TTYPE_SUCCESS`. The function completed successfully.
- `TTYPE_ERROR_INVALID_SOURCEID`. The test script source was incorrectly identified.
- `TTYPE_ERROR`. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Example

```
//*****  
HRESULT TTGetName(const CHAR SourceID[TTYPE_MAX_ID], const CHAR  
ScriptID[TTYPE_MAX_ID], CHAR ScriptName[TTYPE_MAX_NAME], CHAR  
ErrorDescription[TTYPE_MAX_ERROR])  
{  
    AFX_MANAGE_STATE(AfxGetStaticModuleState());  
  
    HRESULT rc = TTYPE_SUCCESS;  
  
    CString sScriptName;  
  
    /* CODE OMITTED: Obtain name of script and store in variable  
       sScriptName.*/  
  
    _tcscopy(ScriptName, sScriptName);  
  
    return rc;  
}
```

See Also

`TTEdit()`, `TTProperties()`

TTGetNode()

Returns information about the node that is identified by the unique ID.

Syntax

```
HRESULT TTGetNode(const TCHAR SourceID[TTYE_MAX_PATH], const
    TCHAR NodeID[TTYE_MAX_ID], struct ScriptNode **pNode, TCHAR
    ErrorDescription[TTYE_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>NodeID</i>	INPUT. The unique ID that identifies the selected test script in the source.
<i>pNode</i>	OUTPUT. A pointer to a populated <i>ScriptNode</i> structure.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TTYE_SUCCESS`. The function completed successfully.
- `TTYE_ERROR`. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

The following structure supports this function.

```
struct ScriptNode
{
    TCHAR    Name [TTYE_MAX_NAME];
    TCHAR    NodeID [TTYE_MAX_ID];
    TCHAR    Type [TTYE_MAX_TYPE];
    BOOL     IsOnlyContainer;
    BOOL     IsImplementation;
};
```

TTGetNode()

Example

```

//*****
TESTTYPEAPI_API HRESULT TTGetNode(const TCHAR SourceID[TTYPE_MAX_ID],
const TCHAR NodeID[TTYPE_MAX_ID], struct ScriptNode** pNode, TCHAR
ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = S_OK;
    *pNode = 0;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TTYPE_ERROR_INVALID_SOURCEID;

    *pNode = new struct ScriptNode;

    // This adapter example is assuming that there is not a node
    // hierarchy.
    _tcsncpy((*pNode)->Name, (const char *)NodeID, TTYPE_MAX_NAME);

    (*pNode)->IsOnlyContainer = FALSE;
    (*pNode)->IsImplementation = TRUE;

    _tscopy((*pNode)->NodeID, NodeID);

    // copy the name of node type into the node structure for all
    // nodes
    _tscopy((*pNode)->Type, pContext->m_sType);

    return rc;
}

```

See Also

TTGetChildren(), TTGetNodeActions(), TTGetSourceControlStatus(),
TTGetTypeIcon()

TTGetNodeActions()

Returns a pointer to an array of test script actions.

Syntax

```
HRESULT TTGetNodeActions(const TCHAR SourceID[TT_MAX_PATH],
    const TCHAR Type[TT_MAX_TYPE], struct Action *pActions[],
    int *pnActionCount, TCHAR ErrorDescription[TT_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test script source.
<i>Type</i>	INPUT. The name of a node type.
<i>pActions</i>	OUTPUT. A local array containing action structures for node <i>Type</i> .
<i>pnActionCount</i>	OUTPUT. The number of actions in <i>pActions</i> .
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TT_SUCCESS. The function completed successfully.
- TT_ERROR. The function did not complete successfully.

Comments

The *Type* parameter is empty if no types have been returned.

See Also

```
TTGetSourceActions(), TTExecuteNodeAction(),
TTExecuteSourceAction()
```

TTGetRoots()

Returns the array of nodes comprising the roots of the test script source.

Syntax

```
HRESULT TTGetRoots(const TCHAR SourceID[TTYE_MAX_PATH], struct
    Node *pRootNodes[], long *plNodeCount, TCHAR
    ErrorDescription[TTYE_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>pRootNodes</i>	OUTPUT. A handle to a window that can be the parent of a dialog displayed by this method.
<i>plNodeCount</i>	OUTPUT. The number of nodes returned by this method.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TTYE_SUCCESS`. The function completed successfully.
- `TTYE_ERROR`. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

The following structure supports this function.

```
struct ScriptNode
{
    CHAR        Name [TTYE_MAX_NAME] ;
    TCHAR       NodeID [TTYE_MAX_ID] ;
    TCHAR       Type [TTYE_MAX_TYPE] ;
    BOOL        IsOnlyContainer;
    BOOL        IsImplementation;
};
```

Example

```

//*****
TESTTYPEAPI_API HRESULT TTGetRoots(const TCHAR SourceID[TTYPE_MAX_ID],
struct ScriptNode *pRootNodes[], long* plNodeCount, TCHAR
ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TTYPE_ERROR_INVALID_SOURCEID;

    // In this example, insert a single folder as a root node
    *pNodeArray = new struct ScriptNode[1];
    *plNodeCount = 1;
    *pChildNodes = 0;

    *pNodeArray[0].IsImplementation = FALSE;
    (*pNodeArray)[0].IsOnlyContainer = TRUE;
    _tcscopy((*pNodeArray)[0].Type, _T("RootFolder"));
    _tcscopy((*pNodeArray)[0].Name, sName);
    _tcscopy((*pNodeArray)[0].NodeID, _T("C:\\RootFolder\\"));

    return rc;
}

```

See Also

TTGetChildren(), TTGetNode(), TTGetSourceControlStatus(),
TTGetTypeIcon()

TTGetSourceActions()

Returns a pointer to an array of actions that can be applied to the test script source.

Syntax

```

HRESULT TTGetSourceActions(const TCHAR
    SourceID[TTYPE_MAX_PATH], struct Action *pActions[], long
    *plActionCount, TCHAR ErrorDescription[TTYPE_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>pActions</i>	OUTPUT. An array of populated Action structures, each of which defines an action.
<i>pIActionCount</i>	OUTPUT. The number of actions returned.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

The adapter returns an array of actions that apply to the test script source.

It is assumed that the action array is allocated by the adapter and deleted by TestManager.

The following structure supports this function:

```
#define TTYPE_ACTION_NAME    100

struct Action
{
    char    Name [TTYPE_ACTION_NAME]:
    int     ActionID;
} ActionType;
```

Example

```

//*****
TESTTYPEAPI_API HRESULT TTGetSourceActions(const TCHAR
SourceID[TTYPE_MAX_ID], struct TTACTION *pActions[], int
*pnActionCount, TCHAR ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;

```



```

CConnectionContext *pContext=0;

// lookup the context information for the specified SourceID
CString sSourceID = SourceID;
m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

if (pContext == 0)
    return TTYPE_ERROR_INVALID_SOURCEID;

*pActions = new TTAction[2];

_tcscpy((*pActions)[0].Name, _T("Custom Action 1"));
((*pActions)[0].ActionID = 1

_tcscpy((*pActions)[1].Name, _T("Custom Action 2"));
((*pActions)[1].ActionID = 2

return rc;
}

```

See Also

[TTExecuteNodeAction\(\)](#), [TTGetNodeActions\(\)](#),
[TTExecuteSourceAction\(\)](#)

TTGetSourceControlStatus()

Returns the current source-control status of the test script.

Syntax

```

HRESULT TTGetSourceControlStatus(const TCHAR
    SourceID[TTYPE_MAX_PATH], const TCHAR NodeID, long
    lWindowContext, TCHAR ErrorDescription[TTYPE_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>NodeID</i>	INPUT. The unique ID that identifies the selected test script in the source.
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this method.

Element	Description
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_CANCEL. The user pressed the Cancel button.
- TTYPE_ERROR_INVALID_SOURCEID. The test script source was incorrectly identified.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Example

```

//*****
TESTTYPEAPI_API HRESULT TTGetSourceControlStatus(const TCHAR
SourceID[TTYPE_MAX_ID], const TCHAR NodeID[TTYPE_MAX_ID], long
*plStatus, TCHAR ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;
    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TTYPE_ERROR_INVALID_SOURCEID;

    try
    {
        long lResult;

        /* CODE OMITTED: Retrieve assets associated with NodeID.
        Retrieve source control status on assets and set lResult
        accordingly. */

        *plStatus = lResult;
    }
    catch (_com_error &e)
    {

```

```

        rc = TTYPE_ERROR;
        CString sError = (BSTR) e.Description();
        tcscopy(ErrorDescription, (LPCTSTR) sError );
    }
    return rc;
}

```

See Also

TTAddToSourceControl(), TTCheckIn(), TTCheckOut(),
TTUndoCheckout()

TTGetSourceIcon()

Returns the path to the icon that represents the test script source.

Syntax

```

HRESULT TTGetSourceIcon(const TCHAR SourceID[TTYPE_MAX_PATH],
    TCHAR IconPath[TTYPE_MAX_PATH], TCHAR
    ErrorDescription[TTYPE_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>IconPath</i>	INPUT. The path to the icon that represents the test script source.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Example

```

//*****
TESTTYPEAPI_API HRESULT TTGetSourceIcon(const TCHAR
SourceID[TTYTYPE_MAX_ID], TCHAR IconPath[TTYTYPE_MAX_PATH], TCHAR
ErrorDescription[TTYTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYTYPE_SUCCESS;

    DWORDdwError;
    CString sIcon;
    char szModuleFileName[_MAX_PATH+1];
    char szModuleFilePath[_MAX_PATH+1];
    char szDir[_MAX_DIR+1];
    char szDrive[_MAX_DRIVE+1];

    // In this sample, the icon is assumed to be in the same location
    // as the adapter
    dwError=GetModuleFileName((HMODULE)AfxGetInstanceHandle(),
szModuleFileName, _MAX_PATH);
    _splitpath(szModuleFileName, szDrive, szDir, NULL, NULL);

    _tcscopy(szModuleFilePath, szDrive);
    _tcscat(szModuleFilePath, szDir);

    sIcon.LoadString(IDS_SOURCE_ICON);
    _tcscat(szModuleFilePath, sIcon);
    _tcscopy(IconPath, szModuleFilePath);

    return rc;
}

```

See Also

TTGetChildren(), TTGetNode(), TTGetIcon(), TTGetTypeIcon()

TTGetTestToolOptions()

Returns the test script options associated with the specified test script.

Syntax

```

HRESULT TTGetTestToolOptions(const TCHAR
    SourceID[TTYTYPE_MAX_PATH], const TCHAR NodeID[TTYTYPE_MAX],
    long lWindowContext, struct ScriptOption *pOptions[], int
    *piCount, TCHAR ErrorDescription[TTYTYPE_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>NodeID</i>	INPUT. The unique ID that identifies the selected test script in the source.
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this method.
<i>pOptions</i>	OUTPUT. An array of adapter-provided test script options.
<i>piCount</i>	OUTPUT. The number of test script options.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TTYPE_SUCCESS`. The function completed successfully.
- `TTYPE_CANCEL`. The user pressed the Cancel button.
- `TTYPE_ERROR`. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.
- `TTYPE_ERROR_INVALID_SOURCEID`. The test script source was incorrectly identified.

Comments

The adapter returns an array name-value pair containing the test script options for the specified test script.

Example

```
//*****
TESTTYPEAPI HRESULT TTGetTestToolOptions(const TCHAR
SourceID[TTYPE_MAX_ID], const TCHAR NodeID[TTYPE_MAX_ID], long
lWindowContext, struct ScriptOption *pOptions[], int* piCount, TCHAR
ErrorDescription[TTYPE_MAX_ERROR]);
    AFX_MANAGE_STATE(AfxGetStaticModuleState())
{
    HRESULT rc = TTYPE_SUCCESS;
```

TTGetTestToolOptions()

```
CConnectionContext *pContext=0;

// lookup the context information for the specified SourceID
CString sSourceID = SourceID;
m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

if (pContext == 0)
    return TTYPE_ERROR_INVALID_SOURCEID;

try
{
    long lResult;

    /* In this example, the username and password are provided as
    connection options. They are needed for the test tool as well,
    but in a slightly different format. */

    CString sUserName;
    CString sPassword;
    // Retrieve the UserID and Password
    for (int iIndex = 0; iIndex < nOptions; iIndex++)
    {
        if (_tcsicmp(pConnectOptions[iIndex].Name, "UserName")
            == 0)
        {
            sUserName = pConnectOptions[iIndex].Value;
        }
        else
            if (_tcsicmp(pConnectOptions[iIndex].Name, "Password")
                == 0)
            {
                sPassword = pConnectOptions[iIndex].Value;
            }
        }

        *pOptions = new ScriptOption[ 2 ];
        *piCount = 2;

        _tcsncpy((*pOptions)[0].Name, _T("USR"));
        _tcsncpy((*pOptions)[0].Value, sUserName);

        _tcsncpy((*pOptions)[1].Name, _T("PSWD"));
        _tcsncpy((*pOptions)[1].Value, sPassword);
    }
catch (_com_error &e)
{
    rc = TTYPE_ERROR;
    CString sError = (BSTR) e.Description();
    tcscopy(ErrorDescription, (LPCTSTR) sError );
}
return rc;
}
```

TTGetTypeIcon()

Returns the path to the bitmap that represents the node type.

Syntax

```
HRESULT TTGetTypeIcon(const TCHAR SourceID[TTYPE_MAX_PATH],
    const TCHAR Type[TTYPE_MAX_TYPE], TCHAR
    IconPath[TTYPE_MAX_PATH], TCHAR
    ErrorDescription[TTYPE_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>Type</i>	INPUT. The type of node for which the icon is being requested.
<i>IconPath</i>	INPUT. The path to the icon that represents the test script type.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

The following structure supports this function.

```
struct ScriptNode
{
    TCHAR    Name [TTYPE_MAX_NAME];
    TCHAR    NodeID [TTYPE_MAX_ID];
    TCHAR    Type [TTYPE_MAX_TYPE];
    BOOL     IsOnlyContainer;
    BOOL     IsImplementation;
};
```

TTNew()

Example

```

//*****
TESTTYPEAPI_API HRESULT TTGetTypeIcon(const TCHAR
SourceID[TTYTYPE_MAX_ID], const TCHAR Type[TTYTYPE_MAX_TYPE], TCHAR
IconPath[TTYTYPE_MAX_PATH], TCHAR ErrorDescription[TTYTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYTYPE_SUCCESS;

    DWORDdwError;
    CString sIcon;
    char szModuleFileName[_MAX_PATH+1];
    char szModuleFilePath[_MAX_PATH+1];
    char szDir[_MAX_DIR+1];
    char szDrive[_MAX_DRIVE+1];

    // In this sample, the icon is assumed to be in the same location
    // as the adapter.
    dwError=GetModuleFileName((HMODULE)AfxGetInstanceHandle(),
szModuleFileName, _MAX_PATH);
    _splitpath(szModuleFileName, szDrive, szDir, NULL, NULL);

    _tcscopy(szModuleFilePath, szDrive);
    _tcscat(szModuleFilePath, szDir);

    sIcon.LoadString(IDS_TYPE_ICON);
    _tcscat(szModuleFilePath, sIcon);
    _tcscopy(IconPath, szModuleFilePath);

    return rc;
}

```

See Also

TTGetChildren(), TTGetNode(), TTGetRoots()

TTNew()

Enables the tester to create a new test script using the hosted tool.

Syntax

```

HRESULT TTNew(const char SourceID[TTYTYPE_MAX_ID], char
ScriptID[TTYTYPE_MAX_ID], char Name[TTYTYPE_MAX_NAME],

```



```
ScriptOption *pScriptOptions[], int* nScriptOptions, char
ErrorDescription[TTYE_MAX_ERROR], long lWindowContext)
```

Element	Description
<i>SourceID</i>	INPUT. A string that identifies the test source.
<i>ScriptID</i>	OUTPUT. A string that identifies the test script.
<i>Name</i>	OUTPUT. A string that specifies the name of a test script.
<i>pScriptOptions</i>	OUTPUT. Reserved for future use.
<i>nScriptOptions</i>	OUTPUT. Reserved for future use.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.
<i>lWindowContext</i>	INPUT. An integer that specifies the handle for the parent window (HWND).

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_ERROR_INVALID_SOURCEID. The test script source was incorrectly identified.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

After the tester creates the new test script, this function passes the ID and test script name to Test Manager. TestManager then passes this information to the `TTedit()` and `TTShowProperties()` functions of this adapter as well as to the Test Script Execution Adapter (TSEA).

Example

```

/*****
HRESULT TTNew(const CHAR SourceID[TTYE_MAX_ID], CHAR ScriptID
[TTYE_MAX_ID], CHAR Name[TTYE_MAX_NAME], ScriptOption
*pScriptOptions[], int* nScriptOptions, CHAR
ErrorDescription[TTYE_MAX_ERROR], long lWindowContext)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

```

TTSelect()

```
HRESULT rc = TTYPE_SUCCESS;

CConnectionContext *pContext=0;

// Lookup the context information for the specified SourceID.
CString sSourceID = SourceID;
m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

// If the context was found, continue.
if (pContext)
{
    /* CODE OMITTED: Create a new Test Script. This could be as
    simple as executing a command line call to your favorite text
    editor or a more complex interaction with a Test Tool.*/

    /* The Name and ID of the new script must be
    returned in the variables ScriptID and ScriptName.*/

    _tcscpy(ScriptID, (const char *)sScriptID);
    _tcscpy(ScriptName, (const char *)sScriptName);
}
else
    rc = TTYPE_ERROR_INVALID_SOURCEID;

return rc;
}
```

See Also

TTEdit(), TTShowProperties()

TTSelect()

Displays a UI that allows a user to select a test script and then returns information that identifies the selected script.

Syntax

```
HRESULT TTSelect(const char SourceID[TTYPE_MAX_ID], char
    ScriptID[TTYPE_MAX_ID], char ScriptName[TTYPE_MAX_ID],
    ScriptOption *pScriptOptions[], int* nScriptOptions, char
    ErrorDescription[TTYPE_MAX_ERROR], long lWindowContext)
```

Element	Description
<i>SourceID</i>	INPUT. A string that identifies the connection.
<i>ScriptID</i>	OUTPUT. A string that identifies the test script.
<i>ScriptName</i>	OUTPUT. A string that specifies the name of a test script.
<i>pScriptOptions</i>	OUTPUT. A structure that specifies options for running the test script.
<i>nOptionOptions</i>	OUTPUT. An integer that specifies the number of test script options.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.
<i>lWindowContext</i>	INPUT. An integer that specifies the handle for the parent window (HWND)

Return Values

This function typically returns one of the following values:

- `TTYPE_SUCCESS`. The function completed successfully.
- `TTYPE_ERROR_INVALID_SOURCEID`. The test script source was incorrectly identified.
- `TTYPE_ERROR`. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

Use this function to provide a UI for the user to select a test script. The function returns the ID and name of the test script to TestManager. These parameters are passed to the `TTEdit()` and `TTShowProperties()` functions and are also passed to the Test Script Execution Adapter (TSEA).

Script options are settings that may be important for the test script to execute properly. The TSCA can specify and modify script options, which are saved by TestManager as part of the test case or suite. The data type `ScriptOption` is defined as follows:

```
struct ScriptOption
{
    char Name[TTYPE_MAX_NAME];
    char Value[TTYPE_MAX_NAME];
}ScriptOptionType;
```

TTSetConfiguration()

Example

```

//*****
TESTTYPEAPI_API HRESULT TTSelect(const char SourceID[TTYPEMAX_ID],
char ScriptID[TTYPER_MAX_ID], char ScriptName[TTYPER_MAX_NAME],
ScriptOption *pScriptOptions[], int *nScriptOptions, char
ErrorDescription[TTYPER_MAX_ERROR], long lWindowContext)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    m_ServerConnections.Lookup(SourceID, (void *)&pContext);

    // if the context was found, then continue
    if (pContext)
    {
        // remove the connection from the list of "live" connections
        m_ServerConnections.RemoveKey(SourceID);
        // Now delete it
        delete pContext;
    }

    return rc;
}

```

See Also

TTEdit(), TTShowProperties()

TTSetConfiguration()

Sets the configuration for the test script source based on the specified configuration buffer.

Syntax

```

HRESULT TTSetConfiguration(const TCHAR
    SourceID[TTYPER_MAX_PATH], TCHAR *pConfigurationBuffer, int
    nConfigurationBufferLength, TCHAR
    ErrorDescription[TTYPER_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>pConfigurationBuffer</i>	INPUT. A pointer to the buffer that contains the streamed configuration.
<i>nConfigurationBufferLength</i>	INPUT. The length of the configuration buffer.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

The adapter sets the configuration for the test script source.

TestManager passes the data returned by a previous call to the method `TTGetConfiguration()` as an input parameter.

Example

```
//*****
TESTTYPEAPI_API HRESULT TTSetConfiguration(const TCHAR
SourceID[TTYPE_MAX_ID], TCHAR *pConfigurationBuffer, int
nConfigurationBufferLength, TCHAR ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TTYPE_ERROR_INVALID_SOURCEID;
}
```

TTSetFilterEx()

```
// Persist the raw filter buffer for possible future use by
// TTGetConfigurationEx or TTCompile, TTEdit, etc...
pContext->m_sConfiguration = pConfigurationBuffer;

/* CODE OMITTED: Handle any configuration application actions
needed by the adapter */

return rc;
}
```

See Also

TTGetConfiguration()

TTSetFilterEx()

Sets the filter for the test script source based on the specified filter buffer.

Syntax

```
HRESULT TTSetFilterEx(const TCHAR SourceID[TTYPE_MAX_PATH],
    TCHAR *pFilterBuffer, int nFilterBufferLength, TCHAR
    ErrorDescription[TTYPE_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>pFilterBuffer</i>	INPUT. A pointer to the buffer that contains the streamed filter.
<i>nFilterBufferLength</i>	INPUT. The length of the filter buffer.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

The adapter sets the filter for the test script source.

TestManager passes, as an input parameter, the data returned by a previous call to the method TTGetFilterEx.

Example

```
//*****
TESTTYPEAPI_API HRESULT TTSetFilterEx(const TCHAR
SourceID[TTYPE_MAX_ID], TCHAR *pFilterBuffer, int nFilterBufferLength,
TCHAR ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TTYPE_ERROR_INVALID_SOURCEID;

    /* The remainder of this code is an example of filtering file
    based script source. In this sample, the TTGetFilterEx returned
    a list of file extensions that were to be hidden from the user.
    This example merely parses the filter buffer and
    stores the results in the adapter's connection context.
    TTGetRoots and TTGetChildren
    would then use this to determine their output. */

    // Persist the raw filter buffer for possible future use by
    // TTGetFilterEx
    pContext->m_sFilter = pFilterBuffer;

    TCHAR szBuffer[512];
    _tcscpy(szBuffer, pFilterBuffer);
    char seps[] = ";";
    char *token;

    pContext->m_asFilterFileExtensions.RemoveAll();

    // establish string and get the first token:
    token = strtok(szBuffer, seps );
    while( token != NULL )
    {
        CString sToken = token;
        sToken.TrimLeft();
        sToken.TrimRight();
    }
}
```

TTShowProperties()

```
        pContext->m_asFilterFileExtensions.Add(sToken);

        // Get next token:
        token = strtok( NULL, seps );

    } // end while

    return rc;
}
```

See Also

TTGetFilterEx()

TTShowProperties()

Displays the properties of a test script.

Syntax

```
HRESULT TTShowProperties(const char SourceID[TTYPE_MAX_ID],
    const char ScriptID[TTYPE_MAX_ID], ScriptOption
    *pScriptOptions[], int *nScriptOptions, char
    ErrorDescription[TTYPE_MAX_ERROR], long lWindowContext)
```

Element	Description
<i>SourceID</i>	INPUT. A string that identifies the connection.
<i>ScriptID</i>	INPUT. A string that identifies the test script.
<i>pScriptOptions</i>	INPUT. The test script options passed in from TTSelect(). Any changes made to the test script options must be passed back and made available to the test case or suite.
<i>nScriptOptions</i>	INPUT. An integer that specifies the number of test script options.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.
<i>lWindowContext</i>	INPUT. An integer that specifies the handle for the parent window (HWND)

Return Values

This function typically returns one of the following values:

- `TTYPE_SUCCESS`. The function completed successfully.
- `TTYPE_ERROR_INVALID_SOURCEID`. The test source was incorrectly identified.
- `TTYPE_ERROR_INVALID_ID`. The adapter could not find a test script with this identification.
- `TTYPE_ERROR`. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Comments

Script options are settings that may be important for the test script to execute properly. The TSCA can specify and modify script options, which are saved by TestManager as part of the test case or suite. The data type `ScriptOption` is defined as follows:

```
struct ScriptOption
{
    char Name[TTYPE_MAX_NAME];
    char Value[TTYPE_MAX_NAME];
}ScriptOptionType;
```

Example

```
//*****
HRESULT TTShowProperties(const CHAR SourceID[TTYPE_MAX_ID],const CHAR
ScriptID [TTYPE_MAX_ID], ScriptOption
*pScriptOptions[], int nScriptOptions, long lWindowContext , char
ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;

    CConnectionContext *pContext=0;

    // Lookup the context information for the specified SourceID.
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    // If the context was found, continue.
    if (pContext)
    {
        /* CODE OMITTED: Display the test script property sheet.*/
    }
}
```

TTUndoCheckout()

```
else
    rc = TTYPE_ERROR_INVALID_SOURCEID;

return rc;
}
```

See Also

TTSelect(), TTEdit()

TTUndoCheckout()

Undoes the checkout of the appropriate files for the specified test script.

Syntax

```
HRESULT TTUndoCheckout(const TCHAR SourceID[TTYPER_MAX_PATH],
    const TCHAR NodeID, long lWindowContext, TCHAR
    ErrorDescription[TTYPER_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The adapter-provided handle that identifies the connection to the test script source.
<i>NodeID</i>	INPUT. The unique ID that identifies the selected test script in the source.
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this method.
<i>ErrorDescription</i>	OUTPUT. A message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_CANCEL. The user pressed the Cancel button.
- TTYPE_ERROR_INVALID_SOURCEID. The test script source was incorrectly identified.
- TTYPE_ERROR. The adapter is using a customized error message. TestManager displays the contents of *ErrorDescription* to the tester.

Example

```

//*****
TESTTYPEAPI_API HRESULT TTUndoCheckout(const TCHAR
SourceID[TTYPE_MAX_ID], const TCHAR NodeID[TTYPE_MAX_ID], long
lWindowContext, TCHAR ErrorDescription[TTYPE_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TTYPE_SUCCESS;
    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TTYPE_ERROR_INVALID_SOURCEID;

    try
    {
        /* CODE OMITTED: Retrieve assets associated with NodeID.
        Perform source control UndoCheckOut on assets. */
    }
    catch (_com_error &e)
    {
        rc = TTYPE_ERROR;
        CString sError = (BSTR) e.Description();
        tcscopy(ErrorDescription, (LPCTSTR) sError );
    }
    return rc;
}

```

See Also

TTAddToSourceControl(), TTCheckIn(), TTCheckOut(),
TTGetSourceControlStatus()

TTUndoCheckout()

Part 2: Adding Custom Test Input Types

Introduction to the Test Input Adapter API

5

This chapter describes the Test Input Adapter (TIA) API and explains how to build a custom TIA.

This chapter contains the following sections:

- About the Test Input Adapter API
- Building a TIA: Workflow and Implementation Issues

About Test Input Adapters

A Test Input Adapter (TIA) is a C or C++ dynamic-link library (DLL) that integrates with TestManager to enable additional test input types to be available for planning test cases. Test cases define what a tester will test, based on this newly-defined test input.

Each test input source is associated with a particular TIA. Without a TIA, the user would be unable access a test input through the user interface.

TIA Functionality

A TIA must, at a minimum:

- Connect to and disconnect from the test input source.
- Return a list of test inputs to TestManager.

In addition to these basic functions, many TIAs are designed to support more sophisticated actions. A more robust TIA might support such operations as:

- Filtering test inputs within the Test Input view.
- Displaying the properties of a test input.
- Performing impact analysis (suspicion) based on changes to test inputs.
- Configuring a test input source.
- Executing a custom action against the test input source or the test input.

Built-In and Custom TIAs

Test inputs consist of any data that the test designer identifies as needing validation. (For more information about built-in test input types, see the *Rational TestManager User's Guide*.)

TestManager comes with three built-in test input types. Built-in TIAs support these test input types, enabling TestManager to access these test inputs. The built-in test input types are:

- Requirements in the Rational RequisitePro project
- Elements in a Rational Rose model
- Rows or columns in a Microsoft Excel spreadsheet

If you want TestManager to support a new test input type, you must implement a custom TIA. That is, you can *extend* TestManager's functionality to support test inputs generated by non-Rational tools. You can define and manage as a test input type any kind of intermediary object needed for testing. Examples are:

- Microsoft Project files
- C++ project files

Each C++ source module associated with a C++ project file can have its own language restrictions, for example, permitted depth of inheritance. You can create test cases that determine whether specific source modules adhere to these restrictions.

Because a TIA must provide connection with the test input source, the following functions are required for building a basic TIA:

- `TICConnectEx()`

(Note that this function supersedes `TICConnect()`, which should only be used if `TICConnectEx()` is not supported.)

- `TIDisconnect()`

In addition, the TIA must provide access to the test inputs in the source by implementing some of the following functions:

- `TIGetRoots()`
- `TIGetNode()`
- `TIGetChildren()`

Other functions are optional; they enable the user to work with the user interface to perform additional operations on the test input. The advantages of using these additional functions are described in *Building a TIA: Workflow and Implementation Issues* on page 185.

Some functions work in pairs. For example, because `TIConnectEx()` is called to make the connection to the test input source, `TIDisconnect()` must subsequently be called to disconnect from the test input source.

For information about specific declarations, see the required header file:

```
...\Rational Test\rtsdk\c\include\testinputapi.h
```

The TIA Function Calls

The TIA applications programming interface (API) consists of 34 functions that are organized into eight functional groupings.

Functional Groupings of TIA Functions

The Test Input Adapter (TIA) functions are summarized in the following table, which shows:

- The functional groupings and their purposes
- The functions within each group

Notes:

- 1 TestManager does not currently call the functions preceded by an asterisk (*).
- 2 The following extended function calls supersede the corresponding non-extended calls:
 - `TIConnectEx()`
 - `TISetFilterEx()`

Function Group and Purpose	Functions in group
Connection Supports connection and disconnection from the test input source.	<code>TIConnect()</code>
	<code>TIConnectEx()</code>
	<code>TIDisconnect()</code>

Function Group and Purpose	Functions in group
Data Access Provides access to the source's test inputs: <ul style="list-style-type: none"> ▪ Hierarchical data ▪ Non-hierarchical data ▪ Sources with different test input types ▪ Functions providing icons for source and test inputs 	TIGetRoots ()
	TIGetChildren ()
	TIGetIsNode ()
	*TIGetParent ()
	*TIGetName ()
	*TIGetType ()
	*TIGetTypes ()
	TIGetTypeIcon ()
	TIGetSourceIcon ()
	TIGetNeedsValidation ()
State Access Provides information about test input states.	*TIGetIsParent ()
	*TIGetIsChild ()
	*TIGetIsNode ()
	*TIGetIsModified ()
	*TIGetIsModifiedSince ()
	*TIGetModified ()
	TIGetModifiedSince ()
Filtering Provides support to filter out test inputs that do not meet user-defined criteria	TISetFilter ()
	TISetFilterEx ()
	TIGetFilterEx ()
	*TISetValidationFilter ()
UI Support Provides the ability to create custom user interface components that display the properties of a test input.	TIShowProperties ()
	*TIShowSelectDialog ()
Source Configuration Support Provides the ability to produce a custom user interface to aid in configuring the test input source	TISetConfiguration ()
	TIGetConfiguration ()

Function Group and Purpose	Functions in group
Custom Action Execution Provides the ability to expose custom operations from TestManager's Test Input view	TIExecuteSourceAction()
	TIExecuteNodeAction()
	TIExecuteSourceAction()
	TIGetNodeActions()
Miscellaneous Infrastructure Provide infrastructure support	*TIGetIsValidSource()
	TIGetIsFunctionSupported()

Mapping of User Actions to TIA Function Calls

A TestManager end user may want to carry out various actions on the test input source. Following is a list of common test input operations that the user might perform in the Test Input view. The order in which these operations are listed represents a plausible sequence in which the user might execute them.

- 1 Defining or modifying the configuration of a test input source.
- 2 Opening the Test Input view.
- 3 Setting a filter for a test input source.
- 4 Selecting a test input from a test input source.
- 5 Displaying properties of a test input.
- 6 Performing custom actions (if any) on the test input source

Note: Based on the current state of TestManager (including which test-input-related operations it has already performed), TestManager may not call some of the functions listed in the following sequences.

Defining or Modifying the Configuration of a Test Input Source

When the user defines or modifies the configuration of a test input source, TestManager calls the following two functions from the **Source Configuration** group in the order listed below.

Typical Sequence of Function Calls	Operation
1 TIGetConfiguration()	Exposes user interface elements that collect data access and data format information from the user when registering the test input source. The TIA passes that information back to TestManager to be persisted as a property of the test input source.
2 TISetConfiguration()	When a connection is made to the test input source, TestManager passes the configuration data obtained from the TIGetConfiguration() function into this function.

Note: Test inputs from different sources demonstrate great variability in:

- Data access
- Data format

For example, test inputs from a RequisitePro project are relatively stable. TestManager treats all requirements in a RequisitePro test input source as potential test inputs. Therefore, you do not need to write a custom TIA to collect any additional user input.

On the other hand, test inputs from an Excel-based input source tend to be highly variable regarding the test input location (column or row) and the data format, making it undesirable to build a specialized adapter for each possible condition.

Opening the Test Input View

When the user opens up the Test Input view, TestManager calls the following functions from the **Connection** group in the order listed below.

Note: TISetConfiguration() supersedes TISetConfiguration().

Typical Sequence of Function Calls Operation

- | | | |
|---|--------------------------------|--|
| 1 | <code>TIConnectEx()</code> | Establishes a connection to the test input source. |
| 2 | <code>TIGetSourceIcon()</code> | Returns the path to the bitmap containing the icon that represents the test input source in Test Input view.

Note that implementation of this function is optional. |

Setting a Filter for a Test Input Source

When the user sets a filter on the test input source, TestManager calls the following functions from the **Filtering** group in the order listed below.

Typical Sequence of Function Calls Operation

- | | | |
|---|------------------------------|---|
| 1 | <code>TIGetFilterEx()</code> | Exposes user interface elements that collect filtering specifications from the user. The TIA passes those filtering specifications back to TestManager to be persisted with the Test Input view or test coverage reports. |
| 2 | <code>TISetFilterEx()</code> | When a connection is made to the test input source, TestManager passes into the TIA the filtering specifications obtained from <code>TIGetFilterEx()</code> . |

Selecting a Test Input from a Test Input Source

Many operations, including associating a test case to a test input, require that the user select a test input from a test input source. To display the contents of a test input source, TestManager calls these functions from the **Data Access** group, typically in the following sequence:

Typical Sequence of Function Calls Operation

- | | | |
|---|--------------------------------|---|
| 1 | <code>TIGetSourceIcon()</code> | Returns the path to the bitmap that represents the test input source. |
| 2 | <code>TIGetRoots()</code> | Returns the array of nodes comprising the roots of the test input source. |

Typical Sequence of Function Calls Operation

3	<code>TIGetChildren()</code>	Returns an array of nodes that are the children of the specified node. TestManager calls this function once for each test input returned by the function <code>TIGetRoots()</code> .
4	<code>TIGetTypeIcon()</code>	Returns the path to the bitmap that represents the test input type for each test input added to the view.

Displaying Properties of a Test Input

When the user makes a UI selection to view the properties of a test input, TestManager calls the following function, which comes from the **UI Support** group.

Function Call	Operation
<code>TIShowProperties()</code>	Displays the properties of a selected test input.

Performing Custom Actions on the Test Input Source

You can implement the TIA to support custom actions on the test input source. If the TIA supports these custom actions, TestManager displays them in a custom menu so that the user can select them. To enable display and execution of custom actions on the test input source, TestManager calls these functions from the **Custom Action Execution** group, typically in the following sequence:

Typical Sequence of Function Calls	Operation	
1	<code>TIGetSourceActions()</code>	Returns a pointer to an array of actions that can be applied to the test input source.
2	<code>TIEecuteSourceAction()</code>	Executes the specified action on the test input source.

Performing Custom Actions on a Test Input Node

You can implement the TIA to support custom actions on a test input node. If the TIA supports these custom actions, TestManager displays them in a custom menu so that the user can select them. To enable display and execution of custom actions on the test input node, TestManager calls these functions from the **Custom Action Execution** group, typically in the following sequence:

Typical Sequence of Function Calls	Operation
1 TIGetNodeActions()	Returns a pointer to an array of actions that can be applied to a node of a particular type.
2 TIExecuteNodeAction()	Executes the specified action on the specified node.

Building a Custom Test Input Adapter

This section first describes the skills you need to build a custom TIA and then describes the implementation issues that arise when you create this adapter.

Prerequisite Skills

To build a custom TIA using TestManager's C/C++ API, you need the following skills:

- A working knowledge of the tool or format used to persist test inputs of that type. This knowledge must include details of how to programmatically access the data within the test input source.
- An ability to build a C or C++ DLL that exposes the functions called by TestManager.
- Familiarity with the TIA API.

Building a TIA: Workflow and Implementation Issues

This section discusses the general workflow that you should follow when building a TIA and the implementation issues that arise at each phase of this workflow. The phases are as follows:

- Making a connection
- Accessing the data
- Supporting impact analysis
- Displaying properties

- Supporting user configuration of the Test Input source
- Filtering
- Custom action support

Making a Connection

The most critical phase in developing a TIA is determining how to use the functions in the Data Access group to access the data (that is, the test inputs stored in a source) and return this data to TestManager. In most cases, the data is accessed using a common API or directly by accessing the physical representation of the data. This physical representation can be file, a database, or some other source.

It is most efficient if the TIA maintains an active connection to the test input source rather than reconnecting each time the user makes a request from TestManager. It is possible that some test input sources do not require a connection to access the test inputs contained within it.

Given these considerations, you, the TIA developer, must determine the answers to the following connection issues at the beginning, before you create the TIA. The decisions you make regarding the connection issues are the most critical decisions you make in the process of building the TIA. Once you have determined satisfactory solutions to the following issues, you are likely to be successful in your development of the TIA.

- 1 How can the TIA gain access to the test input data in the source? If an existing API performs this function, use it. If an appropriate API does not exist, is there a way to gain access to the data directly? Alternatively, is there a consistent way to export the data to CSV or some other common format for which an adapter exists or can be written?
- 2 How can the adapter uniquely identify the test input source? If the test inputs are stored in a file, the preferred form of identification is the path to the file. If the test inputs are not stored in a file, you must create a logical name. The test input source identification that you develop is passed to the `TICConnectEx()` function when TestManager calls it.
- 3 Is any additional information needed for connecting to the source? If not, the `TICConnect()` function will probably be adequate. If, however, additional data is needed, the user can specify it as connection options when registering the test input source. These connection options are passed to `TICConnectEx()`.

- 4 How can the TIA uniquely identify a test input? If the tool used to create the test input supports the concept of a unique ID for each test input, it is optimal to use it. If not, the adapter needs to derive a unique identifier. Each test input needs its own unique ID per source, which is used as a key for association with a test case.
- 5 Does the test input source require any configuration data in addition to the data needed for connection? If so, what data is needed? The major decision you face here is which data to hard code into the TIA itself and which data to have the user specify.

For example, if you are building a general TIA to support general database access of test inputs, you can either:

- Write a table-specific TIA in which you hard code the table and column names into the TIA itself.
- Write a more general TIA that has functionality enabling users to specify the table and column names each time they register a new test input source.

The first option, while requiring less initial work on your part, is not reusable when the test input configuration changes — for example, a new table is used to store the test input data. In this case, you would have to write a new TIA for the new table.

The second option, in which you write the TIA so that the user can specify the table and column names, is more flexible and reusable. To enable the user to specify this information, you must implement the functions `TISetConfiguration()` and `TIGetConfiguration()`. `TIGetConfiguration()` must provide a UI to collect the configuration information.

Accessing the Data

The next phase in building a TIA is to develop support for the functions in the Data Access group.

Once this support is developed, the TIA should be ready for partial use by TestManager. If a new test input type has been registered to use the TIA, the user should be able to do the following:

- Register and, if necessary, configure a test input source
- Use the Test Input view to view the test inputs in that source
- Associate a test case with any of the test inputs from that source

The issues to resolve in developing support for functions in the Data Access group are:

- 1 What is the organizational structure for the test input data that the TIA will access? Is the data organized in a hierarchical structure or in a flat list?

If the test input data is organized in a flat list, you need the following functions to return data:

- `TIGetRoots()`
- `TIGetNode()`

If the test input data is organized hierarchically, you need the following functions to return data:

- `TIGetRoots()`
- `TIGetNode()`
- `TIGetChildren()`

- 2 Are the items contained in the test input source all of the same type, or does the source support multiple types? A Rational Rose model is an example of a test input source that contains many different types of test inputs.
- 3 Is there an easily recognized bitmap or icon that represents the test input source?

Because Windows users are accustomed to recognizing software components by icons, you should support the function `TIGetSourceIcon()`. This function returns the path of the bitmap that represents the test input source.

- 4 Are there easily recognized bitmaps or icons that represent the different elements that comprise test inputs in the test input source?

Because Windows users are accustomed to recognizing software components by icons, you should support the function `TIGetTypeIcon()`. This function returns the path of the bitmap that represents the node type.

Supporting Impact Analysis

A key TestManager feature is its ability to support impact analysis based on changes to test inputs. For this feature to work, you must implement the function `TIGetModifiedSince()`, which returns a list of test inputs that have been modified since a specified date and time. To determine which test inputs have changed, the TIA needs to access a test input source containing a last-modified date/time for each test input. Without such a source, it is unlikely that you can create a TIA that supports impact analysis.

Displaying Properties

By default, the only test input information displayed by TestManager's user interface is the input name. To enable the user to view other associated data about the test input from the TestManager user interface, you should implement the function `TIShowProperties()`. Try to use the underlying tool's property sheet if it is appropriate for display and can be called by the TIA.

Supporting User Configuration of Test Input Data

In designing a TIA, you need to decide whether to implement the TIA in a way that enables the test designer to specify how the adapter should interpret the test input data. If the test designer will be able to specify data interpretation, you must determine the kind of user interface to provide for specifying the configuration.

You may want to support user configuration of test input data because test inputs from different sources vary in data access, data format, and data stability.

For example, test inputs from a RequisitePro project are relatively stable. The built-in Test Input Adapter is hard coded to understand the configuration of RequisitePro projects. Therefore, the test designer does not need to specify a configuration to the adapter.

In contrast, a test input type can be highly variable. For example, the data configuration in a Microsoft Excel spreadsheet is variable, and therefore the TIA does not intrinsically understand this data configuration, such as whether the data is arranged horizontally or vertically. Given the variable nature of test inputs, you cannot build a specialized adapter for each possible condition.

To enable user configuration of test input data, you should implement the following functions:

- `TIGetConfiguration()`

This function prompts the user with a user interface that collects the information needed to configure the adapter. This function must return that information in a buffer so that TestManager can persist it with the test input source.

- `TISetConfiguration()`

This function enables TestManager to pass into the TIA the test input source configuration information found in the buffer. (The content of this buffer is the same as the configuration information collected from the user.) The TIA needs this information to know how to access the data in the test input source.

In summary, consider the variability in format of the test input source when deciding whether to provide configurability to the user.

Filtering

A test input source can contain an enormous number of test inputs. For example, a RequisitePro project can have 10,000 or more requirements, each of which is a possible test input. In cases like this, you must provide filtering support so that only test inputs meeting certain criteria are displayed to the user in Test Input view or in test case distribution reports. If the test input is created in a tool that supports filtering, you should take advantage of it.

If you implement the functions `TISetFilterEx()` and `TIGetFilterEx()`, TestManager can set and get a filter.

Custom Action Support

TestManager enables you to expose operations in the GUI that are useful for the test designer when working with test inputs. For example, a test designer working with a Rational Requisite Pro project as a test input source would probably want to carry out an operation that opens the Requisite Pro project.

You, the adapter writer, can build the adapter so that the test designer can execute an operation to open Requisite Pro from the GUI. You can also support custom operations for specific test inputs.

To support custom actions against the Test Input source, implement the following functions:

- `TIGetSourceActions()`
- `TIExecuteSourceAction()`

To support custom actions against a test input, implement the following functions:

- `TIGetNodeActions()`
- `TIExecuteNodeAction()`

Registering a New Test Input Adapter

After you have implemented the Test Input Adapter and created the DLL file, do the following:

- Compile the DLL using the `__cdecl*` calling convention.
- Use the following procedure to register the DLL with TestManager.

To register the DLL with TestManager:

- 1 From TestManager click **Tools > Manage > Test Input Types**.
- 2 Click **New** in the Manage Test Inputs dialog box. (If the button is disabled, you do not have write privileges. See the *Using the Rational Administrator* manual or Help.)
- 3 Enter the path to the DLL in the space provided for the adapter.
- 4 Click the **Sources** tab.
- 5 Click **Insert**.
- 6 If a message appears asking if the type should be created, click **Yes**.
- 7 In the New Test Input Source dialog box, enter a name of the source (40 characters maximum).
- 8 Enter a description.
- 9 Select an owner from the list.
- 10 In the **Connect Information** field, enter the connection information required for this test input source.
- 11 Click **OK**.

You can also edit, rename, copy, and delete a test input source from the Manage Test Input Types dialog box.

For detailed information about managing extensible test inputs, see the TestManager online Help.

Test Input Adapter Reference

6

This chapter provides reference material for the test inputs applications programming interface (API), including examples that use code from the Rational RequisitePro adapter. For information about specific declarations, see the following header file:

...Rational Test\rtsdk\c\include\testinputapi.h

Summary of TIA Functions

The Test Inputs Adapter (TIA) functions are summarized in the following table.

Note:

- TestManager does not currently call the functions preceded by an asterisk (*).
- The following extended function calls supersede the corresponding non-extended calls:
 - TIConnectEx()
 - TISetFilterEx()

Function	Definition
TIConnect()	Connects to the test input source.
TIConnectEx()	Connects to the test input source and supports additional parameters for connection options.
TIDisconnect()	Disconnects from the test input source.
TIExecuteNodeAction()	Executes the specified action against the specified test input node.
TIExecuteSourceAction()	Executes the specified action against the test input source
TIGetChildren()	Fills an array with the children of a specified parent node,

Summary of TIA Functions

Function	Definition
TIGetConfiguration()	Returns a pointer to a buffer that contains a configuration for the test input source.
TIGetFilterEx()	Returns a pointer to a filter for the test input source.
TIGetIsFunctionSupported()	Indicates whether a specific function is supported by the adapter for an active connection.
TIGetModifiedSince()	Fills an array with input elements modified since a specified date.
TIGetNeedsValidation()	Determines whether an input element requires validation.
TIGetNode()	Returns information about the specified node.
TIGetNodeActions	Returns a pointer to an array of test input actions.
TIGetRoots()	Fills an array with root elements extracted from the input source.
TIGetSourceActions()	Returns a pointer to an array of actions that can be applied to the test input source.
TIGetSourceIcon()	Points to the location of the 16 x 16 bitmap containing the icon that represents the input source in Test Input view.
TIGetTypeIcon()	Points to the location of the bitmap file of the icon that identifies nodes of a specified type.
TISetConfiguration()	Sets the configuration for the test input source based on the specified configuration buffer.
TISetFilter()	Filters display of test input elements.
TISetFilterEx()	Sets the filter for the test input source based on the specified filter buffer.
TIShowProperties()	Displays the property page or dialog box of an input element.
*TIGetIsChild()	Not currently supported.
*TIGetIsModified()	Not currently supported.

Function	Definition
*TIGetIsModifiedSince()	Determines whether an input element has been modified since a specified date.
*TIGetIsNode()	Determines whether a specified input element exists.
*TIGetIsParent()	Determines whether an input element is a parent node.
*TIGetIsValidSource()	Determines whether a specified input source exists.
*TIGetModified()	Fills an array of structures with input elements that have been modified since the last call to TIGetModified().
*TIGetName()	Extracts the user-readable name of an input element.
*TIGetParent()	Finds the parent node of an input element.
*TIGetType()	Extracts the name of the type of an input element.
*TIGetTypes()	Fills an array with an identifier for each type of input element.
*TISetValidationFilter()	Filters operations according to validation status.
*TIShowSelectDialog()	Displays the selection dialog for choosing elements from the input source.

Using the Type Node Structure

Many of the functions in this API make use of parameters of type `Node`, which is defined as follows:

```
struct Node
{
    char Name[TI_MAX_NAME];
    char NodeID[TI_MAX_ID];
    char Type[TI_MAX_TYPE];
    BOOL IsOnlyContainer;
    BOOL NeedsValidation;
} NodeType;
```

`Name` is the name of the input element displayed in the test input view.

`NodeID` is the unique identifier for this input element, assigned by the adapter.

Type is used for associating this input element with a type (for test inputs that subdivide into types).

If `IsOnlyContainer` is set to `TRUE`, the input element contains other test input elements and cannot have test cases associated with it.

If `NeedsValidation` is set to `TRUE`, the input element needs to be tested.

Note on Memory Allocation

The TIA that you develop is responsible for allocating memory for functions that use pointer-to-pointer parameters, for example `TIGetRoots()`. `TestManager` is responsible for deallocating the memory.

TICConnect()

Connects to the test input source.

Syntax

```
HRESULT TICConnect(const TCHAR ConnectInfo[TI_MAX_PATH], const  
TCHAR UserID[TI_MAX_ID], TCHAR SourceID[TI_MAX_ID], TCHAR  
ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>ConnectInfo</i>	INPUT. A string that specifies the location of the test input source, often defined as a path.
<i>UserID</i>	INPUT. A string that identifies the current tester.
<i>SourceID</i>	OUTPUT. A string that identifies the input source. The ID is used in subsequent calls to the adapter.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the <code>TestManager</code> user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_UNABLE_TO_CONNECT`. No connection with the input source was possible.
- `TI_ERROR_INVALID_CONNECTINFO`. The adapter was unable to use the connection information.

Comments

This call has been superseded by `TIConnectEx()`.

After the connection to an input source has been established, the TIA assigns a unique identifier *SourceID*, which can be any string of characters. TestManager uses this identifier for subsequent calls to the adapter. Be sure to document the format of this string. This is particularly important when there are multiple, simultaneous connections.

Example

```

/*****
HRESULT TIConnect(const TCHAR ConnectInfo[TI_MAX_PATH], const TCHAR
UserID[TI_MAX_ID], TCHAR SourceID[TI_MAX_ID], TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    // Look in the connection map to determine whether a connection with
    // the specified RQS file is already established.
    CConnectionContext *pContext=0;
    m_ProjectConnections.Lookup(ConnectInfo, (void *)&pContext);

    // If there is no active connection for the specified ReqPro
    // Project, attempt to connect.
    if (!pContext)
    {
        CFileStatus FileStatus;
        // Determine whether a ReqPro RQS project file exists.
        if (CFile::GetStatus(ConnectInfo, FileStatus) == TRUE)
        {
            try
            {

                /* CODE OMITTED: Establish a connection to the ReqPro
                project using the ReqPro COM Server.*/
            }
        }
    }
}

```

TIConnect()

```
        // If the connection was successful, add the new connection
        // context to the connection map.
        m_ProjectConnections.SetAt(ConnectInfo, pContext);

        // Use the ConnectionInfo as the SourceID.
        _tcscopy(SourceID, ConnectInfo);
    }
    catch (_com_error &e)
    {
        // If ReqPro COM Server throws an exception, return the
        // error using a built in error processing routine.
        PopulateErrorDescription(IDS_ERROR_UNABLE_TO_CONNECT,
            e.WCode(), e.ErrorMessage(), ErrorDescription);
    }
    else
    {
        // The RQS file does not exist, return the appropriate error
        // code.
        rc = TI_ERROR_INVALID_CONNECTINFO;
    }
}
// The connection already exists.
else
{
    _tcscopy(SourceID, ConnectInfo);
}

return rc;
}
```

See Also

TIConnectEx(), TIDisconnect()

TIConnectEx()

Creates a connection to a test input source. This method supersedes `TIConnect()`; it supports additional parameters for connection options.

Syntax

```
HRESULT TIConnectEx(const TCHAR ConnectInfo[TI_MAX_PATH], const
    TCHAR UserID[TI_MAX_ID], const TIConnectOption
    *pConnectOptions, int nOptions, TCHAR SourceID[TI_MAX_ID],
    TCHAR ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>ConnectInfo</i>	INPUT. A string that specifies the location of test inputs, often defined as a path.
<i>UserID</i>	INPUT. A string that identifies the A string that identifies the current tester.
<i>pConnectOption</i>	INPUT. An array of test input source connection options that the user defined using TestManager.
<i>nOptions</i>	INPUT. The number of connection options.
<i>SourceID</i>	OUTPUT. A string that identifies the input source. The ID is used in subsequent calls to the adapter.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_UNABLE_TO_CONNECT`. No connection with the input source was possible.
- `TI_ERROR_INVALID_CONNECTINFO`. The adapter was unable to use the connection information.

Comments

The TIA calls this function before calling any other function. After the connection to an input source has been established, the TIA assigns a unique identifier *SourceID*, which can be any string of characters. TestManager uses this identifier for subsequent calls to the adapter. *SourceID* must remain valid until `TIDisconnect()` is called.

Example

```

//*****
HRESULT TIConnectEx(const char ConnectInfo[TI_MAX_PATH], const char
UserID[TI_MAX_ID], const struct TICconnectOption *pConnectOptions, int
nOptions, char SourceID[TI_MAX_ID], char
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    // Look in the connection map to determine whether a connection
    // with the specified RQS file is already established.
    CConnectionContext *pContext=0;
    m_ProjectConnections.Lookup(ConnectInfo, (void *)&pContext);

    // If there is no active connection for the specified ReqPro
    // Project, attempt to connect.
    if (!pContext)
    {
        CFileStatus FileStatus;
        // Determine whether a ReqPro RQS project file exists.
        if (CFile::GetStatus(ConnectInfo, FileStatus) == TRUE)
        {
            try
            {
                // If ReqPro used connection options, which it does not
                for (int i=0; i<nOptionCount; i++)
                {
                    // Then process the connection options
                }

                /* CODE OMITTED: Establish a connection to the ReqPro
                project using the ReqPro COM Server.*/

                // If the connection was successful, add the new connection
                // context to the connection map.
                m_ProjectConnections.SetAt(ConnectInfo, pContext);

                // Use the ConnectionInfo as the SourceID.
                _tcsncpy(SourceID, ConnectInfo);
            }
            catch (_com_error &e)
            {

```

```

        // If ReqPro COM Server throws an exception, return the
        // error using a built in error processing routine.
        PopulateErrorDescription(IDS_ERROR_UNABLE_TO_CONNECT,
        e.WCode(), e.ErrorMessage(), ErrorDescription);
    }
}
else
{
    // The RQS file does not exist, return the appropriate error
    // code.
    rc = TI_ERROR_INVALID_CONNECTINFO;
}
}
// The connection already exists.
else
{
    _tcscpy(SourceID, ConnectInfo);
}
}
return rc;
}

```

See Also

TIConnect(), TIDisconnect()

TIDisconnect()

Disconnects from the test input source.

Syntax

```

HRESULT TIDisconnect(const TCHAR SourceID[TI_MAX_ID], TCHAR
    ErrorDescription[TI_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

TIDisconnect()

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The specified source information was not correct.
- `TI_ERROR_UNABLE_TO_DISCONNECT`. There was no existing connection to disconnect from.

Comments

After TestManager calls `TIDisconnect()`, no further calls to this input source are allowed without another call to `TICConnect()` or `TICConnectEx()`.

Example

```
//*****
HRESULT TIDisconnect(const TCHAR SourceID[TI_MAX_ID], TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    // Look in the connection map to determine whether a connection with
    // the specified RQS file is already established. Note that the
    // ReqPro adapter also uses the path of the RQS file as the
    // SourceID.

    CConnectionContext *pContext=0;
    m_ProjectConnections.Lookup(ConnectInfo, (void *)&pContext);

    // If the context was found, then continue.
    if (pContext)
    {
        /* CODE OMITTED: Close connection to the ReqPro Project by using
        the ReqPro COM Server.*/

        // Remove the connection from the list of active connections.
        m_ProjectConnections.RemoveKey(SourceID);
    }
    return rc;
}
```

See Also

`TICConnect()`, `TICConnectEx()`

TIExecuteNodeAction()

Executes the specified action against the specified test input node.

Syntax

```
HRESULT TIExecuteNodeAction(const TCHAR SourceID[TI_MAX_PATH],
    const TCHAR Node[TI_MAX_ID], int nActionID, long
    lWindowContext, TCHAR ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>Node</i>	OUTPUT. The ID of the
<i>nActionID</i>	INPUT. The ID of the action to be executed.
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this function.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR`. The function did not complete successfully.

Example

```

/*****
HRESULT TIExecuteNodeAction(const TCHAR SourceID[TI_MAX_ID], const
TCHAR Node[TI_MAX_ID], int nActionID, long lWindowContext, TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID

```

TIExecuteSourceAction()

```
CString sSourceID = SourceID;
m_ServerConnections.Lookup (sSourceID, (void *)&pContext);

if (pContext == 0)
    return TI_ERROR_INVALID_SOURCEID;

switch (nActionID)
{
    case 0:
        /* CODE OMITED: Execute custom action 1. */
        break;
    case 1:
        /* CODE OMITED: Execute custom action 2. */
        break;

    default:
        rc = TI_ERROR;
        _tcscpy(*ErrorDescription,
            _T("Unrecognized action received"));
        break;
}
return rc;
}
```

See Also

TIGetSourceActions(), TIGetNodeActions(),
TIExecuteSourceAction()

TIExecuteSourceAction()

Executes the specified action against the test input source.

Syntax

```
HRESULT TIExecuteSourceAction(const TCHAR
    SourceID[TI_MAX_PATH], int nActionID, long lWindowContext,
    TCHAR ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>nActionID</i>	INPUT. The ID of the action to be executed.
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this function.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR`. The function did not complete successfully.

Example

```

/*****
HRESULT TIExecuteSourceAction(const TCHAR SourceID[TI_MAX_ID], int
nActionID, long lWindowContext, TCHAR ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup (sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TI_ERROR_INVALID_SOURCEID;

    switch (nActionID)
    {
        case 0:
            /* CODE OMITED: Execute custom action 1. */
            break;
        case 1:
            /* CODE OMITED: Execute custom action 2. */
            break;
    }
}

```

TIGetChildren()

```
        default:
            rc = TI_ERROR;
            _tcscpy(*ErrorDescription,
                _T("Unrecognized action received"));
            break;
    }
    return rc;
}
```

See Also

TIGetSourceActions(), TIGetNodeActions(), TIExecuteNodeAction()

TIGetChildren()

Fills an array with the children of a specified parent node.

Syntax

```
HRESULT TIGetChildren(const TCHAR SourceID[TI_MAX_ID], const
    TCHAR NodeID[TI_MAX_ID], struct Node *pChildNodes[], long
    *plNodeCount, TCHAR ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>NodeID</i>	INPUT. A string that identifies a parent node.
<i>pChildNodes</i>	OUTPUT. A pointer to a structure containing the child elements of parent <i>NodeID</i> .
<i>plNodeCount</i>	OUTPUT. A pointer to the number of child elements in <i>pChildNodes</i> .
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

For the definition of type Node, see “Using the Type Node Structure.”

Return Values

This function typically returns TI_SUCCESS when the function completes successfully and returns TI_ERROR_INVALID_SOURCEID if the input source was incorrectly identified.

Comments

This function fills *pChildNodes* with child elements of a parent node specified by *NodeID*. If the parent has no children, *pChildNodes* is empty.

You assign the total number of elements written to *pChildNodes* to *plNodeCount*.

Example

```
//*****
HRESULT TIGetChildren(const TCHAR SourceID[TI_MAX_ID], const TCHAR
NodeID[TI_MAX_ID], struct Node *pChildNodes[], long* plNodeCount,
TCHAR ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    struct Node *pNodeArray=0;
    *plNodeCount = 0;
    *pChildNodes = 0;
    CConnectionContext *pContext=0;

    // Lookup the connection information for the ReqPro Project
    // identified by SourceID.
    m_ProjectConnections.Lookup(SourceID, (void *)&pContext);

    // Determine whether a connection exists with the specified
    // SourceID.
    if (pContext)
    {
        CPtrArray PtrArray;

        /* CODE OMITTED: Obtain a collection of child requirements using
        ReqPro COM server and store the data for each child requirement
        in an instance of class CReqInfo. CReqInfo is a ReqPro adapter
        specific class which stores info about each ReqPro
        requirement. The instances of CReqInfo are stored in a
        pointer array (CPtrArray).*/

        // Allocate enough Node data structures for all the children in
        // the point array.
        pNodeArray = new struct Node[PtrArray.GetSize()];

        // Populate the array of Nodes with the data returned by using
        // the ReqPro COM server.
        for (long lIndex=0; lIndex < PtrArray.GetSize(); lIndex++)
        {
            // Get an instance of CReqInfo.
            CReqInfo *pReqInfo = (CReqInfo *)PtrArray.GetAt(lIndex);

            // Copy the requirement name.
            _tcscpy(pNodeArray[lIndex].Name, (const char *)
```

TIGetChildren()

```
pReqInfo->m_sName);

// Copy the Container and NeedsValidation attributes.
pNodeArray[lIndex].IsOnlyContainer = pReqInfo->m_bContainer;
pNodeArray[lIndex].NeedsValidation =
pReqInfo->m_bNeedsValidation;

// Copy the Requirement NodeID (which is a GUID for a ReqPro
// requirement).
_tcscpy(pNodeArray[lIndex].NodeID, pReqInfo->m_sGUID);

char szNodeType[TI_MAX_TYPE+1];

/* CODE OMITTED: Obtain the name of the requirement type by using
the ReqPro COM server.*/

// Copy the name of requirement type into the node structure.
_tcscpy(pNodeArray[lIndex].Type, (char *) szNodeType);

delete pReqInfo;
}

// Set the return pointer for the node array.
*pChildNodes = pNodeArray;

// Set the count for the number of returned nodes.
*plNodeCount = PtrArray.GetSize();
}
else
    rc = TI_ERROR_INVALID_SOURCEID;

return rc;
}
```

See Also

TIGetParent()

TIGetConfiguration()

Returns a pointer to a buffer that contains a configuration for the test input source.

Syntax

```
HRESULT TIGetConfiguration(const TCHAR SourceID[TI_MAX_PATH],
    long lWindowContext, TCHAR **pConfigurationBuffer, int
    *pnConfigurationBufferLength, TCHAR ErrorDescription)
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>lWindowContext</i>	INPUT. A string that identifies an input element.
<i>pConfigurationBuffer</i>	OUTPUT. A pointer to the buffer that contains the streamed configuration.
<i>pnConfigurationBufferLength</i>	OUTPUT. The length of the configuration buffer.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- **TI_SUCCESS**. The function completed successfully.
- **TI_ERROR**. The function did not complete successfully.
- **TI_ERROR_INVALID_SOURCEID**. The input source was identified incorrectly.

Comments

Typically, the adapter displays a user interface to collect the configuration data.

TestManager is responsible for persisting the data.

Example

```

//*****
HRESULT TIGetConfiguration(const TCHAR SourceID[TI_MAX_ID], long
lWindowContext, TCHAR **pConfigurationBuffer, int*
pnConfigurationBufferLength, TCHAR ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TI_ERROR_INVALID_SOURCEID;

    *pFilterBuffer = new char[_MAX_PATH];

    CWnd ParentWnd;
    ParentWnd.FromHandle((HWND)lWindowContext);

    /* You have to create a class (A dialog with an embedded
    list control to perform configuration selection is used here) */
    CSelectConfigDialog Dialog(pContext, &ParentWnd);

    if (Dialog.DoModal() == IDOK)
    {
        _tcscpy(*pConfigurationBuffer,
            Dialog.m_sSelectedConfigurationName);
        *pnConfigurationBufferLength =
            Dialog.m_sSelectedConfigurationName.GetLength();
    }
    else
    {
        // Put in blanks to indicate no filter set
        CString sEmpty;
        _tcscpy(*pConfigurationBuffer, sEmpty);
        *pnConfigurationBufferLength = sEmpty.GetLength();
    }

    return rc;
}

```

See Also

TISetConfiguration()

TIGetFilterEx()

Returns a pointer to a buffer that contains a filter for the test input source.

Syntax

```
HRESULT TIGetFilterEx(const TCHAR SourceID[TI_MAX_PATH],
    long lWindowContext, TCHAR **pFilterBuffer, int
    *pnFilterBufferLength, TCHAR ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>lWindowContext</i>	INPUT. A handle to a window that can be the parent of a dialog displayed by this function.
<i>pFilterBuffer</i>	OUTPUT. A pointer to the buffer that contains the streamed filter.
<i>pnFilterBufferLength</i>	OUTPUT. The length of the filter buffer.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR`. The function did not complete successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.
-

Comments

The filter is returned as a stream of characters.

The data is interpreted only by the adapter.

Typically, the adapter displays a user interface to collect the filter data.

TestManager is responsible for persisting the data.

TIGetFilterEx()

Example

```
//*****
HRESULT TIGetFilterEx (const TCHAR SourceID[TI_MAX_PATH], long
lWindowContext, TCHAR **pFilterBuffer, int *pnFilterBufferLength,
TCHAR ErrorDescription[TI_MAX_ERROR] )
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    HRESULT rc = TI_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TI_ERROR_INVALID_SOURCEID;

    /* The rest of this code simply displays a dialog that shows the
    filter options. It assumes that the dialog saves the filter
    settings to the Context pointer. */
    if (pContext->m_lpDispatchProject)
    {
        try
        {
            CFilterDialog FilterDialog(pContext, NULL);
            if (FilterDialog.DoModal() == IDOK)
            {
                *pnFilterBufferLength =
                pContext->GetFilterSettings(pFilterBuffer);
            }
            else
                rc = TI_ERROR;
            FilterDialog.DestroyWindow();
        }
        catch (_com_error)
        {
            rc = TI_ERROR;
        }
    }
    else
        rc = TI_ERROR_INVALID_SOURCEID;
}
return rc;
}
```

See Also

TISetFilterEx()

TIGetIsChild()

Note: This function is not currently called.

Determines whether an input element is a child node.

Syntax

```
HRESULT TIGetIsChild(const TCHAR SourceID[TI_MAX_ID], const
    TCHAR NodeID[TI_MAX_ID], BOOL* pbIsChild, TCHAR
    ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>NodeID</i>	INPUT. A string that identifies an input element.
<i>pbIsChild</i>	OUTPUT. A pointer to a Boolean value that specifies whether the node is a child.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was incorrectly identified.
- `TI_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

Set the Boolean value *pbIsChild* to `TRUE` if the element identified in *NodeID* is a child. Set the value to `FALSE` if the element is not a child.

See Also

`TIGetIsParent()`

TIGetIsFunctionSupported()

Indicates whether a specific function is supported by the adapter for an active connection.

Syntax

```
HRESULT TIGetIsFunctionSupported(const TCHAR
    SourceID[TI_MAX_PATH], long lFunctionID, BOOL *pbSupported,
    TCHAR ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>lFunctionID</i>	INPUT. The constant definition of the function.
<i>pbSupported</i>	OUTPUT. A boolean indicating whether the specified function is supported.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- **TI_SUCCESS**. The function completed successfully.
- **TI_ERROR**. The function did not complete successfully.

Comments

The *lFunctionID* argument can be one of the following:

```

TI_FUNCTION_TIClearFilter
TI_FUNCTION_TICConnect
TI_FUNCTION_TICConnectEx
TI_FUNCTION_TIDisconnect
TI_FUNCTION_TIExecuteNodeAction
TI_FUNCTION_TIExecutionSourceAction
TI_FUNCTION_TIGetChildren
TI_FUNCTION_TIGetConfiguration
TI_FUNCTION_TIGetFilterEx
TI_FUNCTION_TIGetIsChild
TI_FUNCTION_TIGetIsModified
TI_FUNCTION_TIGetIsNode
TI_FUNCTION_TIGetIsParent
TI_FUNCTION_TIGetIsValidSource
TI_FUNCTION_TIGetModified
TI_FUNCTION_TIGetModifiedSince
TI_FUNCTION_TIGetName
TI_FUNCTION_TIGetNeedsValidation
TI_FUNCTION_TIGetNode
TI_FUNCTION_TIGetNodeActions
TI_FUNCTION_TIGetParent
TI_FUNCTION_TIGetRoots
TI_FUNCTION_TIGetSourceActions
TI_FUNCTION_TIGetSourceIcon
TI_FUNCTION_TIGetType
TI_FUNCTION_TIGetTypeIcon
TI_FUNCTION_TIGetTypes
TI_FUNCTION_TISetConfiguration
TI_FUNCTION_TISetFilter
TI_FUNCTION_TISetFilterEx
TI_FUNCTION_TISetValidationFilter
TI_FUNCTION_TIShowProperties
TI_FUNCTION_TIShowSelectDialog

```

Example

```

//*****

HRESULT TIGetIsFunctionSupported(const TCHAR SourceID[TI_MAX_ID],
const TCHAR NodeID[TI_MAX_ID], long lFunctionID, BOOL *pbIsSupported,
TCHAR ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    /* The following is a sample for a minimal adapter with no source
       control support. */

    switch (lFunctionID)
    {

```

TIGetIsModified()

```
        case TI_FUNCTION_TICheck:
        case TI_FUNCTION_TIDisconnect:
        case TI_FUNCTION_TIEdit:
        case TI_FUNCTION_TIGetIcon:
        case TI_FUNCTION_TIGetName:
        case TI_FUNCTION_TINew:
        case TI_FUNCTION_TISelect:
        case TI_FUNCTION_TIShowProperties:
            *pbIsSupported = TRUE;
    break;
    default:
        *pbIsSupported = FALSE;
    break;
}
return TI_SUCCESS;
}
```

TIGetIsModified()

Note: This function is not currently called.

Determines whether an input element has been modified since the last call to `TIGetModified()`.

Syntax

```
HRESULT TIGetIsModified(const TCHAR SourceID[TI_MAX_ID], const
    TCHAR NodeID[TI_MAX_ID], BOOL* pbIsModified, TCHAR
    ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>NodeID</i>	INPUT. A string that identifies an input element.
<i>pbIsModified</i>	OUTPUT. A pointer to a Boolean value that specifies whether the input element has been modified.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.

- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.
- `TI_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

An element is considered modified if certain properties have changed, particularly properties that affect the way the element is associated with or validated by test cases — for example, the element’s type or its user-readable name.

Set the Boolean value *pbIsModified* to `TRUE` if the element has been modified since the last call to `TIGetModified()`, and set the value to `FALSE` if the element has not been modified.

See Also

`TIGetModified()`, `TIGetIsModifiedSince()`

TIGetIsModifiedSince()

Note: This function is not currently called.

Determines whether an input element has been modified since a specified date.

Syntax

```
HRESULT TIGetIsModifiedSince(const TCHAR SourceID[TI_MAX_ID],
    const TCHAR NodeID[TI_MAX_ID], struct tm tmDate, BOOL*
    pbIsModified, TCHAR ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>NodeID</i>	INPUT. A string that identifies an input element.
<i>tmDate</i>	INPUT. The date, defined in <code>time.h</code> , which is part of the C-language Standard Library.
<i>pbIsModified</i>	OUTPUT. A pointer to a Boolean value that specifies whether the input element has been modified.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

TIGetIsNode()

For the definition of type `Node`, see “Using the Type Node Structure.”

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.
- `TI_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

An element is considered modified if certain properties have changed, particularly properties that affect the way the element is associated with or validated by test cases — for example, the element’s type or its user-readable name.

See Also

`TIGetIsModified()`

TIGetIsNode()

Note: This function is not currently called.

Determines whether a specified input element exists.

Syntax

```
HRESULT TIGetIsNode(const TCHAR SourceID[TI_MAX_ID], const  
TCHAR NodeID[TI_MAX_ID], BOOL* pbIsNode, TCHAR  
ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>NodeID</i>	INPUT. A string that identifies an input element.
<i>pbIsNode</i>	OUTPUT. A pointer to a Boolean value that specifies whether the input element exists.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.
- `TI_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

Set the Boolean value *pbIsNode* to `TRUE` if the element is there, and set the value to `FALSE` if the element is not there.

Example

```
//*****
HRESULT TIGetIsNode(const TCHAR SourceID[TI_MAX_ID], const TCHAR
NodeID[TI_MAX_ID], BOOL* pbIsNode, TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;
    *pbIsNode = FALSE;

    // Look in the connection map to determine whether a connection with
    // the specified RQS file exists. Note that the ReqPro adapter also
    // uses the path of the RQS file as the SourceID.

    CConnectionContext *pContext=0;
    m_ProjectConnections.Lookup(ConnectInfo, (void *)&pContext);

    if (pContext)
    {
        /* CODE OMITTED: Determine whether specified NodeID is valid and set
        value of *pbIsNode based on its validity.*/
    }
    else
        rc = TI_ERROR_INVALID_SOURCEID;

    return rc;
}
```

See Also

`TIGetIsChild()`, `TIGetIsParent()`

TIGetIsParent()

Determines whether an input element is a parent node.

Syntax

```
HRESULT TIGetIsParent(const TCHAR SourceID[TI_MAX_ID], const
    TCHAR NodeID[TI_MAX_ID], BOOL* pbIsParent, TCHAR
    ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>NodeID</i>	INPUT. A string that identifies an input element.
<i>pbIsParent</i>	OUTPUT. A pointer to a Boolean value that specifies whether the input element is a parent node in a hierarchical tree.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.
- `TI_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

Set the Boolean value *pbIsParent* to `TRUE` if the specified input element (*NodeID*) is a parent. Set the value to `FALSE` if the input element is not a parent.

Example

```

//*****
HRESULT TIGetIsParent(const TCHAR SourceID[TI_MAX_ID], const TCHAR
NodeID[TI_MAX_ID], BOOL* pIsParent, TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

```

```

HRESULT rc = TI_SUCCESS; //

// Look in the connection map to determine whether a connection with
// the specified RQS file exists. Note that the ReqPro adapter also
// uses the path of the RQS file as the SourceID.
CConnectionContext *pContext=0;
m_ProjectConnections.Lookup(SourceID, (void *)&pContext);

if (pContext)
{

    /* CODE OMITTED: Determine if specified NodeID is a parent and set
    value of *pbIsParent.*/

}
else
    rc = TI_ERROR_INVALID_SOURCEID;

return rc;
}

```

See Also

TIGetIsChild(), TIGetIsNode()

TIGetIsValidSource()

Note: This function is not currently called.

Determines whether the specified test input source exists.

Syntax

BOOL **TIGetIsValidSource**(const TCHAR *SourceID*[TI_MAX_ID])

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.

Return Values

This function returns a value of TRUE if the input source exists and returns a value of FALSE if the input source is not valid.

TIGetModified()

Example

```
/******  
  
BOOL TIGetIsValidSource(const TCHAR SourceID[TI_MAX_ID])  
{  
    AFX_MANAGE_STATE(AfxGetStaticModuleState());  
  
    BOOL bValidSource = FALSE;  
  
    CConnectionContext *pContext=0;  
    m_ProjectConnections.Lookup(SourceID, (void *)&pContext);  
  
    if (pContext)  
        bValidSource = TRUE;  
  
    return bValidSource;  
}
```

See Also

TIConnect(), TIDisconnect()

TIGetModified()

Note: This function is not currently called.

Fills an array of structures with input elements that have been modified since the last call to TIGetModified().

Syntax

```
HRESULT TIGetModified(const TCHAR SourceID[TI_MAX_ID], struct  
    Node ModifiedNodes[], long* pINodeCount, TCHAR  
    ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>ModifiedNodes</i>	OUTPUT. An array containing elements that have been modified since the last invocation of this call.
<i>pINodeCount</i>	OUTPUT. A pointer to a long integer that specifies the number of elements assigned to <i>ModifiedNodes</i> .
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

For the definition of type `Node`, see “Using the Type Node Structure.”

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.

Comments

This function determines which elements have been modified since the last call to `TIGetModified()`. An element is considered modified if certain properties have changed, particularly properties that affect the way the element is associated with or validated by test cases — for example, the element’s type or its user-readable name. Modified elements are assigned to *ModifiedNodes*.

You assign the total number of modified elements to *plNodeCount*.

See Also

`TIGetIsModified()`, `TIGetModifiedSince()`

TIGetModifiedSince()

Fills an array of node structures with input elements modified since a specified date.

Syntax

```
HRESULT TIGetModifiedSince(const TCHAR SourceID[TI_MAX_ID],
    struct tm tmDate, struct Node *pModifiedNodes[], long*
    plNodeCount, TCHAR ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>tmDate</i>	INPUT. The date, defined in <code>time.h</code> , which is part of the C-language Standard Library.
<i>pModifiedNodes</i>	OUTPUT. A pointer to an array containing elements that have been modified since <i>tmDate</i> .

Element	Description
<i>p1NodeCount</i>	OUTPUT. A pointer to a long integer that specifies the total number of elements written to <i>ModifiedNodes</i> .
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

For the definition of type Node, see “Using the Type Node Structure.”

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.

Comments

An element is considered modified if certain properties have changed, particularly properties that affect the way the element is associated with or validated by test cases — for example, the element’s type or its user-readable name. You assign modified elements to *ModifiedNodes*.

You assign the total number of modified elements to *p1NodeCount*.

Example

```
//*****
HRESULT TIGetModifiedSince(const char SourceID[TI_MAX_ID], struct tm
tmDate, struct Node *pModifiedNodes[], long* p1NodeCount, char
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;
    try
    {
        CTIAConnection *pConnection = 0;

        // lookup the context information for the specified SourceID
        if (!theApp.m_ConnectionMap.Lookup(SourceID, (void
            *&)pConnection))q
            return TI_ERROR_INVALID_SOURCEID;

        // if the context was found, then continue
        if (pConnection)
        {
```

```

// Convert the passed in time
COleDateTime DateTime;
DateTime.SetDateTime(tmDate.tm_year+1900,tmDate.tm_mon+1,
    tmDate.tm_mday,tmDate.tm_hour,tmDate.tm_min,tmDate.tm_sec);

/* CODE OMITTED: Process the new date to see if any of the
   input has been modified since this date.
   And populate the NodeArray to return*/
*pModifiedNodes = pNodeArray;
*plNodeCount = PtrArray.GetSize();
}
else
    rc = TI_ERROR;
}
catch(ColeException *e)
{
    sprintf(ErrorDescription, "ColeException. SCODE: %08lx.",
        (long)e->m_sc);
    return TI_ERROR;
}

catch(ColeDispatchException *e)
{
    sprintf(ErrorDescription,
        "ColeDispatchException. SCODE: %08lx,Description: \"%s\".",
        (long)e->m_wCode, (LPSTR)e->m_strDescription.GetBuffer
            (TI_MAX_ERROR));
    return TI_ERROR;
}

catch(...)
{
    return TI_ERROR;
}

return rc;
}

```

See Also

TIGetModified(), TIGetIsModified(), TIGetIsModifiedSince()

TIGetName()

TIGetName()

Note: This function is not currently called.

Extracts the name of an input element.

Syntax

```
HRESULT TIGetName(const TCHAR SourceID[TI_MAX_ID], const TCHAR  
NodeID[TI_MAX_ID], TCHAR Name[TI_MAX_NAME], TCHAR  
ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>NodeID</i>	INPUT. A string that identifies an input element.
<i>Name</i>	OUTPUT. A string that specifies the name of the input element.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.
- `TI_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

You assign the name of the input element identified in *NodeID* to *Name*.

Example

```
/**  
*****  
HRESULT TIGetName(const TCHAR SourceID[TI_MAX_ID], const TCHAR  
NodeID[TI_MAX_ID], TCHAR Name[TI_MAX_NAME], TCHAR  
ErrorDescription[TI_MAX_ERROR])  
{  
    AFX_MANAGE_STATE(AfxGetStaticModuleState());  
  
    HRESULT rc = TI_SUCCESS;
```



```

// Look in the connection map to determine whether a connection with
// the specified RQS file exists.
CConnectionContext *pContext=0;
m_ProjectConnections.Lookup(ConnectInfo, (void *)&pContext);

// Determine whether a connection exists with the specified
// SourceID.
if (pContext)
{
    CString sReqName;

    /* CODE OMITTED: Obtain the name for the requirement whose ID is the
    value of NodeID and store it in local variable sReqName.
    If value of NodeID is not a valid test input, return
    TI_NODE_NOT_FOUND.*/

    // Copy the name into the return buffer.
    _tcsncpy(Name, (const char *)sReqName, TI_MAX_NAME);
}
else
    rc = TI_ERROR_INVALID_SOURCEID;

return rc;
}

```

See Also

TIGetParent(), TIGetNode()

TIGetNeedsValidation()

Note: This function is not currently called.

Determines whether an input element requires validation.

Syntax

```

HRESULT TIGetNeedsValidation(const TCHAR SourceID[TI_MAX_ID],
    const TCHAR NodeID[TI_MAX_ID], BOOL* pbNeedsValidation,
    TCHAR ErrorDescription[TI_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>NodeID</i>	INPUT. A string that identifies an input element.

TIGetNode()

Element	Description
<i>pbNeedsValidation</i>	OUTPUT. A pointer to a Boolean value that specifies whether the input element requires validation.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.
- `TI_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

Set the value of Boolean *pbNeedsValidation* to `TRUE` if the input element needs to be validated, and set the value to `FALSE` if the input element does not need to be validated.

See Also

`TIGetModified()`, `TIGetModifiedSince()`, `TISetValidationFilter()`

TIGetNode()

Returns information about the specified node.

Syntax

```
HRESULT TIGetNode(const TCHAR SourceID[TI_MAX_ID], const TCHAR  
                  NodeID[TI_MAX_ID], struct Node **pNode, TCHAR  
                  ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>NodeID</i>	INPUT. A string that identifies a parent node.

Element	Description
<i>pNode</i>	OUTPUT. A pointer to the node specified in <i>NodeID</i> .
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns `TI_SUCCESS` when the function completes successfully and returns `TI_ERROR_INVALID_SOURCEID` if the input source was incorrectly identified.

Comments

The definition of type `Node` is as follows:

```
struct Node
{
    char Name[TI_MAX_NAME];
    char NodeID[TI_MAX_ID];
    char Type[TI_MAX_TYPE];
    BOOL IsOnlyContainer;
    BOOL NeedsValidation;
} NodeType;
```

Example

```
HRESULT TIGetNode(TCHAR SourceID[TI_MAX_ID], TCHAR NodeID[TI_MAX_ID],
struct Node **pNode, TCHAR ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = S_OK;

    // Look in the connection map to determine if a connection with the
    //specified .RQS file exists.
    CConnectionContext *pContext=0;
    m_ProjectConnections.Lookup(SourceID, (void *)&pContext);

    if (pContext)
    {
        ...
        //CODE OMITTED: Obtain the Requirement identified by NodeID and
        //store data in instance of CReqInfo.
        //CReqInfo is a ReqPro adapter-specific class that stores info
        //about a ReqPro requirement.
        ...
        //If value of NodeID is not a valid TestInput, TI_NODE_NOT_FOUND is
```

TIGetNodeActions()

```
//returned.
...
// populate Node structure
*pNode = new struct Node;

// copy Requirement Name
_tcscpy((*pNode)->Name, (const char *)pReqInfo->m_sName);

(*pNode)->IsOnlyContainer = pReqInfo->m_bContainer;
(*pNode)->NeedsValidation = pReqInfo->m_bNeedsValidation;

// copy the Requirement NodeID (which is a GUID for a ReqPro
// requirement)
_tcscpy((*pNode)->NodeID, pReqInfo->m_sGUID);

TCHAR szNodeType[TI_MAX_TYPE+1];
...
// CODE OMITTED: Obtain the name of the Requirement Type via the
// ReqPro COM server
...

// copy the name of requirement type into the node structure
_tcscpy((*pNode)->Type, (char *) szNodeType);
}
else
    rc = TI_ERROR_INVALID_SOURCEID;

return rc;
}
```

See Also

TIGetIsNode()

TIGetNodeActions()

Returns a pointer to an array of test input actions.

Syntax

```
HRESULT TIGetNodeActions(const TCHAR SourceID[TI_MAX_PATH],
    const TCHAR Type[TI_MAX_TYPE], struct Action *pActions[],
    int *pnActionCount, TCHAR ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>Type</i>	INPUT. The name of a node type.
<i>pActions</i>	OUTPUT. A local array containing action structures for node <i>Type</i> .
<i>pnActionCount</i>	OUTPUT. The number of actions in <i>pActions</i> .
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TI_SUCCESS. The function completed successfully.
- TI_ERROR. The function did not complete successfully.
- TI_ERROR_INVALID_SOURCEID. The input source was identified incorrectly.

Comments

The *Type* parameter is empty if no types have been returned.

Example

```

//*****
HRESULT TIGetNodeActions(const TCHAR SourceID[TI_MAX_ID], const TCHAR
Type[TI_MAX_TYPE], struct TIAction *pActions[], int *pnActionCount,
TCHAR ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TI_ERROR_INVALID_SOURCEID;
}

```

TIGetParent()

```
struct TIAction *pTempActions;
pTempActions = new struct TIAction [2];
if ( Type == "FEAT")
{
    _tcscpy(pTempActions[0].Name, "Custom Action for features 1\0");
    pTempActions[0].ActionID = 0;

    _tcscpy(pTempActions[1].Name, "Custom Action features 2\0");
    pTempActions[1].ActionID = 1;
}
*pActionCount = 2;
*pActions = pTempActions;
return rc;
}
```

See Also

TIGetSourceActions(), TIExecuteNodeAction(),
TIExecuteNodeAction()

TIGetParent()

Note: This function is not currently called.

Finds the parent node of an input element.

Syntax

```
HRESULT TIGetParent(const TCHAR SourceID[TI_MAX_ID], const  
TCHAR NodeID[TI_MAX_ID], struct Node** pParentNode, TCHAR  
ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>NodeID</i>	INPUT. A string that identifies an input element.
<i>pParentNode</i>	OUTPUT. A pointer to a structure that receives the parent node.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

For the definition of type Node, see “Using the Type Node Structure.”

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.
- `TI_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

You assign a pointer to the parent node in *pParentNode*. If there is no parent for the specified input element, assign a null pointer.

Example

```
//*****
HRESULT TIGetParent(const TCHAR SourceID[TI_MAX_ID], const TCHAR
NodeID[TI_MAX_ID], struct Node **pParentNode, TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    // Look in the connection map to determine whether a connection with
    // the specified RQS file exists.
    CConnectionContext *pContext=0;
    m_ProjectConnections.Lookup(SourceID, (void *)&pContext);

    if (pContext)
    {
        /* CODE OMITTED: Obtain the parent of the Requirement identified by
        NodeID and store data in instance of CReqInfo.
        If value of NodeID is not a valid test input,
        TI_NODE_NOT_FOUND is returned.*/

        // Populate Node structure.
        *pParentNode = new struct Node;

        // Copy test input name.
        _tcscpy((*pParentNode)->Name, (const char *)pReqInfo->m_sName);

        (*pParentNode)->IsOnlyContainer = pReqInfo->m_bContainer;
        (*pParentNode)->NeedsValidation = pReqInfo->m_bNeedsValidation;

        // Copy the Requirement NodeID (which is a GUID for a ReqPro
        // requirement).
        _tcscpy((*pParentNode)->NodeID, pReqInfo->m_sGUID);
    }
}
```

TIGetRoots()

```
char szNodeType[TI_MAX_TYPE+1];

/* CODE OMITTED: Obtain the name of the requirement type by using
the ReqPro COM server.*/

// Copy the name of requirement type into the node structure.
_tcscpy((*pParentNode)->Type, (char *) szNodeType);
}
else
rc = TI_ERROR_INVALID_SOURCEID;

return rc;
}
```

See Also

TIGetIsNode(), TIGetIsChild()

TIGetRoots()

Fills an array with root elements extracted from the input source.

Syntax

```
HRESULT TIGetRoots(const TCHAR SourceID[TI_MAX_ID], struct Node


*pRootNodes[], long *pNodeCount, TCHAR
ErrorDescription[TI_MAX_ERROR])


```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>pRootNodes</i>	OUTPUT. A string of structures of type Node that receives the root nodes.
<i>pNodeCount</i>	OUTPUT. A pointer to a long integer that specifies the total number of root nodes.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

For the definition of type Node, see “Using the Type Node Structure.”

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.
- `TI_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

You assign nodes that are root elements to *pRootNodes*. If the input source is not hierarchical, fill *RootNodes* with all of the elements of the input source.

You assign *pNodeCount* the total number of nodes written out to *pRootNodes*.

Example

```

//*****
HRESULT TIGetRoots(const TCHAR SourceID[TI_MAX_ID], struct Node
*pRootNodes[], long* pNodeCount, TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    struct Node *pNodeArray=0;
    *pNodeCount = 0;
    *pRootNodes = 0;
    CConnectionContext *pContext=0;

    // Lookup the connection information for the ReqPro Project
    // identified by SourceID.
    m_ProjectConnections.Lookup(SourceID, (void *)&pContext);

    // Determine whether a connection exists with the specified
    // SourceID.
    if (pContext)
    {
        CPtrArray PtrArray;

        /* CODE OMITTED: Obtain a collection of root nodes using ReqPro
        COM server and store the data for Requirement in an instance of
        class CReqInfo. CReqInfo is a ReqPro adapter specific class
        that stores info about each ReqPro requirement. The instances
        of CReqInfo are stored in a pointer array (CPtrArray).*/

        // Allocate enough Node data structures for all the root nodes in
        // the pointer array.

```

TIGetRoots()

```
pNodeArray = new struct Node[PtrArray.GetSize()];

// Populate the array of Nodes with the data returned by using
// the ReqPro COM server.
for (long lIndex=0; lIndex < PtrArray.GetSize(); lIndex++)
{
    // Get an instance of CReqInfo.
    CReqInfo *pReqInfo = (CReqInfo *)PtrArray.GetAt(lIndex);

    // Copy the requirement name.
    _tcscpy(pNodeArray[lIndex].Name, (const char
    *)pReqInfo->m_sName);

    // Copy the Container and NeedsValidation attributes.
    pNodeArray[lIndex].IsOnlyContainer = pReqInfo->m_bContainer;
    pNodeArray[lIndex].NeedsValidation =
        pReqInfo->m_bNeedsValidation;

    // Copy the Requirement NodeID (which is a GUID for a ReqPro
    // requirement).
    _tcscpy(pNodeArray[lIndex].NodeID, pReqInfo->m_sGUID);

    char szNodeType[TI_MAX_TYPE+1];

    /* CODE OMITTED: Obtain the name of the requirement type by using
    the ReqPro COM server.*/

    // Copy the name of requirement type into the node structure.
    _tcscpy(pNodeArray[lIndex].Type, (char *) szNodeType);

    delete pReqInfo;
}

// Set the return pointer for the node array.
* pRootNodes = pNodeArray;

// Set the count for the number of returned nodes.
*plNodeCount = PtrArray.GetSize();
}
else
    rc = TI_ERROR_INVALID_SOURCEID;

return rc;
}
```

See Also

TIGetIsNode(), TIGetIsParent(), TIGetParent(), TIGetChildren()

TIGetSourceActions()

Returns a pointer to an array of actions that can be applied to the test input source.

Syntax

```
HRESULT TIGetSourceActions(const TCHAR SourceID[TI_MAX_PATH],
    struct Action *pActions[], int *pnActionCount, TCHAR
    ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>pActions</i>	OUTPUT. An array of populated action structures, each of which defines an action.
<i>pnActionCount</i>	OUTPUT. The number of actions returned
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TI_SUCCESS. The function completed successfully.
- TI_ERROR. The function did not complete successfully.

Example

```
//*****
HRESULT TIGetSourceActions(const TCHAR SourceID[TI_MAX_ID], struct
    TIAction *pActions[], int *pnActionCount, TCHAR
    ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);
```

TIGetSourceIcon()

```
    if (pContext == 0)
        return TI_ERROR_INVALID_SOURCEID;

    struct TIAction *pTempActions;
    pTempActions = new struct TIAction [2];

    _tcscpy(pTempActions[0].Name, "Custom Action 1\0");
    pTempActions[0].ActionID = 0;

    _tcscpy(pTempActions[1].Name, "Custom Action 2\0");
    pTempActions[1].ActionID = 1;

    *pActionCount = 2;
    *pActions = pTempActions;
    return rc;
}
```

See Also

TIGetNodeActions(), TIExecuteSourceAction(),
TIExecuteNodeAction()

TIGetSourceIcon()

Points to the location of the 16 x 16 bitmap containing the icon that is displayed with the name of the test input source in the TestManager Test Input view.

Syntax

HRESULT **TIGetSourceIcon**(const TCHAR *SourceID*[TI_MAX_ID], TCHAR
IconPath[TI_MAX_PATH], TCHAR *ErrorDescription*[TI_MAX_ERROR])

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>IconPath</i>	OUTPUT. A string that identifies the location of the icon graphics file.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was incorrectly identified.

Comments

The source is identified in *SourceId*. This icon is also used for all elements of the input source that belong to a type that does not have its own icon.

Example

```
//*****
HRESULT TIGetSourceIcon(const TCHAR SourceID[TI_MAX_ID], TCHAR
IconPath[TI_MAX_PATH], TCHAR ErrorDescription[TI_MAX_ERROR])
{
    DWORDdwError;
    charszModuleFileName[_MAX_PATH+1];
    charszModuleFilePath[_MAX_PATH+1];
    charszDir[_MAX_DIR+1];
    charszDrive[_MAX_DRIVE+1];

    // Obtain the path to where the ReqPro adapter is installed.
    dwError=GetModuleFileName((HMODULE)AfxGetInstanceHandle(),
    szModuleFileName, _MAX_PATH);
    _splitpath(szModuleFileName, szDrive, szDir, NULL, NULL);

    // Build the path to where the bitmap file exists.
    _tcscopy(szModuleFilePath, szDrive);
    _tcscat(szModuleFilePath, szDir);
    _tcscat(szModuleFilePath, "bitmap_source.bmp");

    // Copy the path into the return variable.
    _tcscopy(IconPath, szModuleFilePath);

    return TI_SUCCESS;
}
```

See Also

`TIGetTypeIcon()`

TIGetType()

Note: This function is not currently called.

Extracts the name of the type of an input element.

Syntax

```
HRESULT TIGetType(const TCHAR SourceID[TI_MAX_ID], const TCHAR
    NodeID[TI_MAX_ID], TCHAR Type[TI_MAX_TYPE], TCHAR
    ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>NodeID</i>	INPUT. A string that identifies an input element.
<i>Type</i>	OUTPUT. A string that identifies the type of node.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.
- `TI_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

You assign the type of the input element identified in *NodeID* to *Type*.

Example

```

/*****
HRESULT TIGetType(const TCHAR SourceID[TI_MAX_ID], const TCHAR
NodeID[TI_MAX_ID], TCHAR Type[TI_MAX_NAME], TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

```

```

// Look in the connection map to determine whether a connection with
// the specified RQS file exists.
CConnectionContext *pContext=0;
m_ProjectConnections.Lookup(ConnectInfo, (void *)&pContext);

// Determine whether a connection exists with the specified
// SourceID.
if (pContext)
{
    CString sReqType;

    /* CODE OMITTED: Obtain the type for the requirement whose unique ID
    is the value of NodeID and store it in local variable sReqType.
    If value of NodeID is not a valid test input, return
    TI_NODE_NOT_FOUND.*/

    // Copy its type into the return buffer.
    _tcsncpy(Type, (const char *)sReqType, TI_MAX_TYPE);
}
else
    rc = TI_ERROR_INVALID_SOURCEID;

return rc;
}

```

See Also

TIGetTypeIcon(), TIGetTypes()

TIGetTypeIcon()

Points to the location of the 16 x 16 bitmap file of the icon that identifies nodes of a specified type.

Syntax

```

HRESULT TIGetTypeIcon(const TCHAR SourceID[TI_MAX_ID], const
    TCHAR Type[TI_MAX_TYPE], TCHAR IconPath[TI_MAX_PATH], TCHAR
    ErrorDescription[TI_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>Type</i>	INPUT. A string that identifies the type of node.

TIGetTypeIcon()

Element	Description
<i>IconPath</i>	OUTPUT. A string that specifies the location of a graphics file for an icon that represents nodes of a particular type.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns `TI_SUCCESS` when the function completes successfully and returns `TI_ERROR_INVALID_SOURCEID` if the input source information was incorrectly identified.

Comments

Use *IconPath* to point to the location of the icon that identifies nodes of type *Type*.

Example

```
//*****  
HRESULT TIGetTypeIcon(const TCHAR SourceID[TI_MAX_ID], const TCHAR  
Type[TI_MAX_TYPE], TCHAR IconPath[TI_MAX_PATH], TCHAR  
ErrorDescription[TI_MAX_ERROR])  
{  
    DWORD dwError;  
    char szModuleFileName[_MAX_PATH+1];  
    char szModuleFilePath[_MAX_PATH+1];  
    char szDir[_MAX_DIR+1];  
    char szDrive[_MAX_DRIVE+1];  
  
    // Obtain the path to where the ReqPro adapter is installed.  
    dwError=GetModuleFileName((HMODULE)AfxGetInstanceHandle(),  
    szModuleFileName, _MAX_PATH);  
    _splitpath(szModuleFileName, szDrive, szDir, NULL, NULL);  
  
    // Build the path to where the bitmap file exists.  
    _tcscpy(szModuleFilePath, szDrive);  
    _tcscat(szModuleFilePath, szDir);  
    _tcscat(szModuleFilePath, "bitmap_type.bmp");  
  
    // Copy the path into the return variable.  
    _tcscpy(IconPath, szModuleFilePath);  
  
    return TI_SUCCESS;  
}
```


See Also

TIGetSourceIcon()

TIGetTypes()

Note: This function is not currently called.

Fills an array with an identifier for each type of input element.

Syntax

```
HRESULT TIGetTypes(const TCHAR SourceID[TI_MAX_ID], TCHAR
    (**Types) [TI_MAX_TYPE], long* pITypeCount, TCHAR
    ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>Types</i>	OUTPUT. A pointer to an array of type identifiers.
<i>pITypeCount</i>	OUTPUT. A pointer to a long integer that specifies the total number of test types.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns `TI_SUCCESS` when the function completes successfully and returns `TI_ERROR_INVALID_SOURCEID` if the input source information was incorrectly identified.

Comments

You assign an identifier for each input type to *Types*.

You assign the total count of type identifiers to *pITypeCount*.

TIGetTypes()

Example

```

//*****
HRESULT TIGetTypes(const TCHAR SourceID[TI_MAX_ID], TCHAR (**
Types)[TI_MAX_TYPE], long* pTypeCount, TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    // Lookup the connection information for the ReqPro Project
    // identified by SourceID.
    CConnectionContext *pContext=0;
    m_ProjectConnections.Lookup(SourceID, (void *)&pContext);

    // Determine whether a connection exists with the specified
    // SourceID.
    if (pContext)
    {
        try
        {
            // Get the ReqPro project interface pointer from the instance
            // of CConnectionContext.
            ReqServer::_ProjectPtr
            ProjectPtr(pContext->m_lpDispatchProject);

            // Get the requirement types from the ReqPro project interface
            // pointer.
            ReqServer::_ReqTypesPtr
            MyReqTypesPtr(ProjectPtr->GetReqTypes());

            // Allocate enough memory to return the types - (which are
            // strings in ReqPro).
            *Types = (char *) [TI_MAX_TYPE] malloc(sizeof(char) *
            MyReqTypesPtr->GetCount() * TI_MAX_TYPE);

            // Loop through the requirements types to find the right one.
            for (long lIndex = 1; lIndex <= MyReqTypesPtr->GetCount();
                lIndex++)
            {
                ColeVariant varIndex=lIndex;

                /* CODE OMITTED: Obtain requirement type from collection and
                store in variable MyReqTypePtr.*/

                // Copy name of requirement type into the return array. Note
                // that index of array is 0 based.
                _tcsncpy((*pszTypes)[lIndex-1], (char
                *)MyReqTypePtr->GetName());
            }
            // Store count of types in return variable.
            *pTypeCount = MyReqTypesPtr->GetCount();
        }
    }
}

```

```

        catch (_com_error)
        {
            rc = TI_ERROR;
        }
    }
    else
        rc = TI_ERROR_INVALID_SOURCEID;

    return rc;
}

```

See Also

TIGetType(), TIGetTypeIcon()

TISetConfiguration()

Sets the configuration for the test input source based on the specified configuration buffer.

Syntax

```

HRESULT TISetConfiguration(const TCHAR SourceID[TI_MAX_PATH],
    TCHAR *pConfigurationBuffer, int nConfigurationBufferLength,
    TCHAR ErrorDescription[TI_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>pConfigurationBuffer</i>	INPUT. A pointer to the buffer that contains the streamed configuration.
<i>nConfigurationBufferLength</i>	INPUT. The length of the configuration buffer.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR`. The function did not complete successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.

TISetFilter()

Example

```

//*****
HRESULT TISetConfiguration(const TCHAR SourceID[TI_MAX_ID], TCHAR
*pConfigurationBuffer, int nConfigurationBufferLength, TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TI_ERROR_INVALID_SOURCEID;

    // Persist the raw filter buffer for possible future use by
    //TIGetConfigurationEx or
    // TICompile, TIEdit, etc...
    pContext->m_sConfiguration = pConfigurationBuffer;

    /* CODE OMITED: Handle any configuration application actions
       needed by the adapter */

    return rc;
}

```

See Also

TIGetConfiguration()

TISetFilter()

Filters the display of test input elements.

Syntax

```

HRESULT TISetFilter(const TCHAR SourceID[TI_MAX_ID], const
TCHAR UserID[TI_MAX_ID], TCHAR
ErrorDescription[TI_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.

Element	Description
<i>UserID</i>	INPUT. A string that identifies the current user.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was incorrectly identified.

Comments

You provide the filter creation mechanism in a window or in a dialog box so that the display of input elements is reduced to a specific set. Filtering information can be stored on a per-user basis.

Example

```

//*****
HRESULT TISetFilter(const TCHAR SourceID[TI_MAX_ID], const TCHAR
UserID[TI_MAX_ID], TCHAR ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    // Lookup the connection information for the ReqPro Project
    // identified by SourceID.
    CConnectionContext *pContext=0;
    m_ProjectConnections.Lookup(SourceID, (void *)&pContext);

    // Determine whether a connection exists with the specified
    // SourceID.
    if (pContext)
    {
        try
        {
            CFilterDialog FilterDialog(pContext, NULL);

            // Display filter dialog.
            if (FilterDialog.DoModal())
            {
            }
        }
    }
    catch (_com_error)

```

```

TISetFilterEx()
{
    {
    }
}
else
    rc = TI_ERROR_INVALID_SOURCEID;

return rc;
}

```

See Also

TISetFilterEx(), TISetValidationFilter()

TISetFilterEx()

Sets the filter for the test input source based on the specified filter buffer.

Syntax

```

HRESULT TISetFilterEx(const TCHAR SourceID[TI_MAX_PATH], TCHAR
    *pFilterBuffer, int nFilterBufferLength, TCHAR
    ErrorDescription[TI_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>pFilterBuffer</i>	OUTPUT. A pointer to a buffer that contains the streamed filter.
<i>nFilterBufferLength</i>	OUTPUT. The length of the filter buffer.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TI_SUCCESS. The function completed successfully.
- TI_ERROR. The function did not complete successfully.

Comments

This function supersedes the function `TISetFilter()`.

TestManager is responsible for persisting this data.

Example

```

//*****
HRESULT TISetFilterEx(const TCHAR SourceID[TI_MAX_ID], TCHAR
*pFilterBuffer, int nFilterBufferLength, TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    CConnectionContext *pContext=0;

    // lookup the context information for the specified SourceID
    CString sSourceID = SourceID;
    m_ServerConnections.Lookup(sSourceID, (void *)&pContext);

    if (pContext == 0)
        return TI_ERROR_INVALID_SOURCEID;

    /* The remainder of this code is an example of filtering file
    based input source. In this sample, the TIGetFilterEx returned
    a list of file extensions that were to be hidden from the user.
    This example merely parses the filter buffer and stores the
    results in the adapter's connection context. TIGetRoots and
    TIGetChildren would then use this to determine their output.*/

    // Persist the raw filter buffer for possible future use
    // by TIGetFilterEx
    pContext->m_sFilter = pFilterBuffer;

    TCHAR szBuffer[512];
    _tcscpy(szBuffer, pFilterBuffer);
    char seps[] = ";";
    char *token;

    pContext->m_asFilterFileExtensions.RemoveAll();

    // establish string and get the first token:
    token = strtok(szBuffer, seps );
    while( token != NULL )
    {
        CString sToken = token;
        sToken.TrimLeft();
        sToken.TrimRight();
        pContext->m_asFilterFileExtensions.Add(sToken);
    }
}

```

TISetValidationFilter()

```
        // Get next token:
        token = strtok( NULL, seps );

    } // end while

    return rc;
}
```

See Also

TIGetFilterEx()

TISetValidationFilter()

Note: This function is not currently called.

Filters operations according to validation status.

Syntax

```
HRESULT TISetValidationFilter(const TCHAR SourceID[TI_MAX_ID],
    const TCHAR UserID[TI_MAX_ID], BOOL bOnlyNeedsValidation,
    TCHAR ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>UserID</i>	INPUT. A string that identifies the current user.
<i>bOnlyNeedsValidation</i>	INPUT. A pointer to a Boolean value that specifies whether filtering by validation status is in effect.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- **TI_SUCCESS.** The function completed successfully.
- **TI_ERROR_INVALID_SOURCEID.** The input source was incorrectly identified.

Comments

Create a filter so that all operations on the input source (*SourceID*) performed by the current tester (*UserID*) apply only to elements that need validation. If *bOnlyNeedsValidation* is set to `False`, filtering by this criterion is disabled.

See Also

`TISetFilter()`, `TISetFilterEx()`, `TIGetFilterEx()`

TIShowProperties()

Displays the property page or dialog box of an input element.

Syntax

```
HRESULT TIShowProperties(TCHAR const SourceID[TI_MAX_ID], const
    TCHAR NodeID[TI_MAX_ID], struct Node** pModifiedNode, TCHAR
    ErrorDescription[TI_MAX_ERROR])
```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>NodeID</i>	INPUT. A string that identifies an input element.
<i>pModifiedNode</i>	OUTPUT. A pointer to a structure that receives the new information about a node.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCEID`. The input source was identified incorrectly.
- `TI_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

TIShowProperties()

Comments

If any property relevant to TestManager has changed, assign the new information to a Node structure associated with the pointer *pModifiedNode*.

Example

```
//*****
HRESULT TIShowProperties(const TCHAR SourceID[TI_MAX_ID], const TCHAR
NodeID[TI_MAX_ID], struct Node** pModifiedNode, TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    // Look in the connection map to determine whether a connection with
    // the specified RQS file exists.
    CConnectionContext *pContext=0;
    m_ProjectConnections.Lookup(SourceID, (void *)&pContext);

    if (pContext)
    {
        /* CODE OMITTED: Display the ReqPro Requirement Properties dialog
        If OK is selected, store relevant information in an instance of
        CReqInfo. CReqInfo is a ReqPro adapter-specific class.
        If value of NodeID is not a valid test input,
        TI_NODE_NOT_FOUND is returned.*/

        // Populate Node structure.
        *pNode = new struct Node;

        // Copy test input name.
        _tcscpy((*pModifiedNode)->Name, (const char *)pReqInfo->m_sName);

        (*pModifiedNode)->IsOnlyContainer = pReqInfo->m_bContainer;
        (*pModifiedNode)->NeedsValidation =
        pReqInfo->m_bNeedsValidation;

        // Copy the Requirement NodeID (which is a GUID for a ReqPro
        // requirement).
        _tcscpy((*pModifiedNode)->NodeID, pReqInfo->m_sGUID);

        char szNodeType[TI_MAX_TYPE+1];

        /* CODE OMITTED: Obtain the name of the requirement type by using
        the ReqPro COM server.*/

        // Copy the name of requirement type into the node structure.
        _tcscpy((*pModifiedNode)->Type, (char *) szNodeType);
    }
    else
        rc = TI_ERROR_INVALID_SOURCEID;
}
```

```

    return rc;
}

```

See Also

TIShowSelectDialog()

TIShowSelectDialog()

Note: This function is not currently called.

Displays a selection dialog box for choosing elements from the input source.

Syntax

```

HRESULT TIShowSelectDialog(const TCHAR SourceID[TI_MAX_ID],
    struct Node* pSelectedNodes[], long* plNodeCount, TCHAR
    ErrorDescription[TI_MAX_ERROR])

```

Element	Description
<i>SourceID</i>	INPUT. The handle identifying the connection to the test input source.
<i>pSelectedNodes</i>	OUTPUT. An array of structures that specifies the selected input elements.
<i>plNodeCount</i>	OUTPUT. A pointer to a long integer that specifies the total number of nodes selected.
<i>ErrorDescription</i>	OUTPUT. The message to be displayed to the TestManager user if there is an error.

Return Values

This function typically returns one of the following values:

- TI_SUCCESS. The function completed successfully.
- TI_ERROR_INVALID_SOURCEID. The input source was incorrectly identified.

Comments

This function displays a selection dialog box (supplied by the adapter) for selecting elements from the input source (*SourceID*). You assign the selected elements to *pSelectedNodes*.

Assign the total number of selected elements to *p1NodeCount*.

Example

```

/*****
HRESULT TIShowSelectDialog(const TCHAR SourceID[TI_MAX_ID], struct
Node *pSelectedNodes[], long* p1NodeCount, TCHAR
ErrorDescription[TI_MAX_ERROR])
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    HRESULT rc = TI_SUCCESS;

    struct Node *pNodeArray=0;
    // Look in the connection map to determine whether a connection with
    // the specified RQS file exists.
    CConnectionContext *pContext=0;
    m_ProjectConnections.Lookup(SourceID, (void *)&pContext);

    // Determine whether a connection exists with the specified
    // SourceID.
    if (pContext)
    {
        CPtrArray PtrArray;

        /* CODE OMITTED: Display the RequisitePro Requirement Selection
        Dialog. Upon selection of the OK button, extract the selected
        requirements and store the data for each requirement in an
        instance of class CReqInfo.
        CReqInfo is a ReqPro adapter-specific class that stores info
        about each ReqPro requirement. The instances of CReqInfo are
        stored in a pointer array (CPtrArray).*/

        // Allocate enough Node data structures for all the root nodes in
        // the pointer array.
        pNodeArray = new struct Node[PtrArray.GetSize()];

        // Populate the array of Nodes with the data returned by using
        // the ReqPro COM server.
        for (long lIndex=0; lIndex < PtrArray.GetSize(); lIndex++)
        {
            // Get instance of CReqInfo.
            CReqInfo *pReqInfo = (CReqInfo *)PtrArray.GetAt(lIndex);

            // Copy the requirement name.
            _tcscpy(pNodeArray[lIndex].Name, (const char

```

```

*)pReqInfo->m_sName);

// Copy the Container and NeedsValidation attributes.
pNodeArray[lIndex].IsOnlyContainer = pReqInfo->m_bContainer;
pNodeArray[lIndex].NeedsValidation =
pReqInfo->m_bNeedsValidation;

// Copy the Requirement NodeID (which is a GUID for a ReqPro
// requirement).
_tcscpy(pNodeArray[lIndex].NodeID, pReqInfo->m_sGUID);

char szNodeType[TI_MAX_TYPE+1];

/* CODE OMITTED: Obtain the name of the requirement type by
using the ReqPro COM server.*/

// Copy the name of requirement type into the node structure.
_tcscpy(pNodeArray[lIndex].Type, (char *) szNodeType);

delete pReqInfo;
}

// Set the return pointer for the node array.
*pSelectedNodes = pNodeArray;

// Set the count for the number of returned nodes.
*pNodeCount = PtrArray.GetSize();
}
else
    rc = TI_ERROR_INVALID_SOURCEID;

return rc;
}

```

See Also

TIShowProperties()

TIShowSelectDialog()

Index

A

- add
 - command line test script type 3
 - custom test script type 9
 - test script types 2
- advanced
 - list of functions 101
- application
 - get process id 58
 - start 59
 - start (Java) 71
 - wait for termination id 60
- attributes
 - of computers 63
 - of test cases 67, 68

B

- block on shared variable 90

C

- calculate think-time 104
- C/C++ 107
- close
 - datapool 27
 - session 18
 - task 21
- Command Line test script type 2
- command line TSCA 108
- command runtime status, report 82
- command timer
 - start 47
 - stop 45
- command, log 102

- command-line execution engine 12
- computer configuration attribute list, get 63
- computer configuration attribute value, get 64
- configuration attributes
 - of computers 63
 - of test cases 67, 68
- connecting to
 - TSS server 95
- context information, pass to TSS server 96
- create
 - command line test script type 3
 - custom test script type 9
 - task 21
 - test script type 2
- custom test script adapter, building 116

D

- datapools
 - access order during playback 31
 - close 27
 - get column name 28
 - get column value 37
 - get number of columns 27
 - get number of rows 34
 - list of functions 26
 - open 30
 - overview 26
 - reset access 33, 36
 - rewind 33
 - search for column/value pair 35
 - set row access 29
- definition of TSCA 107
- delay script execution 61
- disconnect from TSS server 98
- DLL, registering for TSCA 172

E

- environment variables
 - set 48
- errors
 - get details 62
 - print message 75
- event log 39

F

- functional groupings of TSCA functions 109
- functions, summary of 122

G

- get
 - application process id 58
 - computer configuration attribute list 63
 - computer configuration attribute value 64
 - elapsed runtime 50
 - error details 62
 - exponentially distributed random number 72
 - internal variable value 51
 - name of datapool column 28
 - number of datapool columns 27
 - number of datapool rows 34
 - pathname 65
 - random number 73
 - run state 83
 - script option 66
 - script source file position 79
 - session option 18
 - task option value 22
 - test case configuration 69
 - test case configuration attribute list 68
 - test case configuration attribute value 67
 - test case name 70
 - test tool execution option 70
 - uniformly distributed random number 76
 - unique text string 77
 - value of datapool column 37
 - value of shared variable 89

H

- header file, TSCA 111, 122, 179
- header files
 - TIA 193
 - TSCA 111

I

- internal variables
 - get value of 51
 - list 52
 - set value of 101

J

- Java test script type 2

L

- log
 - command 102
 - event 39
 - message 41
 - test case result 42
- logging
 - list of functions 38, 39, 44, 57

M

- Manual test script type 2
- mapping of user actions to TSCA function
 - calls 111
- measurement
 - list of functions 44, 57
- memory allocation 196
- message
 - log 41
 - print 76
- monitor
 - list of functions 78
- monitor display message, set 79

O

open

- datapool 30
- session 19
- task 21

option

- get session option 18
- get task option value 22
- set session option 20
- set task option value 23

P

pathname, get 65

print

- error message 75
- message 76

proxy TSS server

- start 98
- stop 99

proxy TSS server process

- pass context information to 96

R

random numbers

- get 73
- get (exponentially distributed) 72
- get (uniform) 76
- seed 74

report, command runtime status 82

reset

- datapool access 33, 36

rewind

- datapool 33

rttsee 12

rttsee.exe 10

rttss.dll 10

rttss.h 10

rttssremote.dll 10

run states

- get 83
- list of 84
- set 83

S

script option, get 66

script types 1

search

- datapool 35

seed

- random number generator 74

session

- list of functions 94

SessionClose 18

SessionGetOption 18

SessionOpen 19

SessionSetOption 20

set

- command timer start point 47

- command timer stop point 45

- datapool row access 29

- environment variable 48

- monitor display message 79

- run state 83

- script execution delay 61

- script source file position 81

- session option 20

- synchronization point 93

- task option 23

- think-time delay 53

- timer end point 55

- timer start point 54

- value of internal variable 101

- value of shared variable 87

shared variables

- assignment operations 88

- block on 90

- get value of 89

- set value of 87

- SQABasic test script type 2
- stand-alone TSS server process
 - pass context information to 96
 - start 98
 - stop 99
- start
 - application 59
 - command timer 47
 - Java application 71
 - timer 54
 - TSS server process 98
- stop
 - command timer 45
 - timer 55
 - TSS server process 99
- synchronization
 - list of functions 87
- synchronization point
 - set 93

T

- tables
 - TIA functions 193
- TaskAbort 20
- TaskClose 21
- TaskCreate 21
- TaskExecute 22
- TaskGetOption 22
- TaskSetOption 23
- TEdit
 - example 134
- test case
 - get configuration 69
 - get name 70
 - log result 42
- test case configuration attribute list, get 68
- test case configuration attribute value, get 67
- test script adapter, components 10
- Test Script Console Adapter. *See* TSCA
- test script types 1
 - adding 2, 3, 9

- test scripts
 - block on shared variable 90
 - built-in and custom 108
 - get line position 79
 - get shared variable value 89
 - set line position 81
 - set shared variable value 87
 - set synchronization point 93
- test tool option, get 70
- testtypeapi.h 10
- think time
 - calculate 104
 - set 53
- TIA examples
 - TIConnect 197
 - TIConnectEx 200
 - TIDisconnect 202
 - TIExecuteNodeAction 203
 - TIExecuteSourceAction 205
 - TIGetChildren 207
 - TIGetConfiguration 210
 - TIGetFilterEx 212
 - TIGetIsFunctionSupported 215
 - TIGetIsNode 219
 - TIGetIsParent 220
 - TIGetIsValidSource 222
 - TIGetModifiedSince 224
 - TIGetName 226
 - TIGetNode 229
 - TIGetNodeActions 231
 - TIGetParent 233
 - TIGetRoots 235
 - TIGetSourceActions 237
 - TIGetSourceIcon 239
 - TIGetType 240
 - TIGetTypeIcon 242
 - TIGetTypes 244
 - TISetConfiguration 246
 - TISetFilter 247
 - TISetFilterEx 249
 - TIShowProperties 252
 - TIShowSelectDialog 254
- TIA function table 193

- TIA functions
 - TICConnect 125, 127, 136, 142, 144, 151, 153, 162, 168, 170, 173, 196, 204, 206, 212, 232, 246, 250
 - TICConnectEx 199
 - TIDisconnect 201
 - TIExecuteNodeAction 203
 - TIExecuteSourceAction 204
 - TIGetChildren 203, 213
 - TIGetConfiguration 209
 - TIGetFilterEx 211
 - TIGetIsChild 213
 - TIGetIsFunctionSupported 214
 - TIGetIsModified 216
 - TIGetIsModifiedSince 217
 - TIGetIsNode 218
 - TIGetIsParent 220
 - TIGetIsValidSource 221
 - TIGetModified() 222
 - TIGetModifiedSince 223
 - TIGetName 226
 - example 226
 - TIGetNeedsValidation 227
 - TIGetNodeActions 238
 - TIGetParent
 - example 233
 - TIGetRoots 234
 - example 235
 - TIGetSourceActions 237
 - TIGetSourceIcon
 - example 239
 - TIGetType 240
 - example 240
 - TIGetTypeIcon 241
 - example 242
 - TIGetTypes 243
 - example 244
 - timer
 - calculate think-time 104
 - get elapsed runtime 50
 - set think time 53
 - start 47, 54
 - stop 45, 55
 - TISetFilter 246
 - example 247
 - TISetValidationFilter 250
 - TIShowProperties 251
 - example 252
 - TIShowSelectDialog 253
 - example 254
- TIA functionsTIGetParent 232
- TICConnect 125, 127, 136, 142, 144, 151, 153, 162, 168, 170, 173, 196, 204, 206, 212, 232, 246, 250
- example 197
- TICConnectEx 199
- TIDisconnect 201
- example 202
- TIGetChildren 203, 213
- example 207
- TIGetIsChild 213
- TIGetIsFunctionSupported 214
- TIGetIsModifiedSince 217
- TIGetIsNode 218
- example 219
- TIGetIsParent 220
- example 220
- TIGetIsValidSource 221
- TIGetModified() 222
- TIGetModifiedSince 223
- TIGetName 226
- example 226
- TIGetNeedsValidation 227
- TIGetNodeActions 238
- TIGetParent
- example 233
- TIGetRoots 234
- example 235
- TIGetSourceActions 237
- TIGetSourceIcon
- example 239
- TIGetType 240
- example 240
- TIGetTypeIcon 241
- example 242
- TIGetTypes 243
- example 244
- timer
 - calculate think-time 104
 - get elapsed runtime 50
 - set think time 53
 - start 47, 54
 - stop 45, 55
- TISetFilter 246
- example 247
- TISetValidationFilter 250
- TIShowProperties 251
- example 252
- TIShowSelectDialog 253
- example 254
- TSCA
 - building a custom test script adapter 116
 - built-in and custom test script types 108
 - command line TSCA 108

- custom TSCA 108
- defining or modifying the configuration of a
 - test script source 112
- definition 107
- editing test script properties 114
- editing test script text 114
- functional groupings 109
- header file 111, 122, 179
- integrating with source control 114
- issues in building
 - accessing the data 117
 - custom action support 120
 - displaying properties 119
 - filtering 120
 - integration with source control 119
 - making a connection 116
 - supporting user configuration of the test
 - script source 119
- mapping of user actions to function calls 111
- opening the test script view 112
- performing custom actions on the test script
 - or the test script source 114
- registering the TSCA DLL with
 - TestManager 121
- required and optional functionality 110
- required functionality 107
- selecting a test script for operations 113
- setting a filter for test scripts 113
- summary of functions 122
- TSCA (Test Script Console Adapter)
 - about 107, 177
- TSCA examples
 - TTConnect 129
 - TTDisconnect 132
 - TTEdit 134
 - TTGetIcon 144
 - TTGetName 148
 - TTNew 163
 - TTSelect 166
 - TTShowProperties 171
- TSCA functions
 - TTAddToSourceControl 122
 - TTCheckIn 124
 - TTCheckOut 125
 - TTClearFilter 127
 - TTConnect 128
 - TTDisconnect 132
 - TTEdit 133
 - TTExecuteNodeAction 135
 - TTExecuteSourceAction 136
 - TTGetChildren 138
 - TTGetConfiguration 140
 - TTGetFilterEx 142
 - TTGetIcon 144
 - TTGetIsFunctionSupported 145
 - TTGetName 147
 - TTGetNode 149
 - TTGetNodeActions 151
 - TTGetRoots 152
 - TTGetSourceActions 153
 - TTGetSourceControlStatus 155
 - TTGetSourceIcon 157
 - TTGetTestToolOptions 158
 - TTGetTypeIcon 161
 - TTNew 162
 - TTSelect 164
 - TTSetConfiguration 166
 - TTSetFilterEx 168
 - TTShowProperties 170
 - TTUndoCheckout 172
- TSEError 24
- tsea.h 10
- TSS server process
 - connect to 95
 - disconnect from 98
 - pass context information to 96
 - start 98
 - stop 99
- TSSApplicationPid 58
- TSSApplicationStart 59
- TSSApplicationWait 60
- TSSCommandEnd 45
- TSSCommandStart 47
- TSSConnect 95
- TSSContext 96
- TSSDatapoolClose 27
- TSSDatapoolColumnCount 27
- TSSDatapoolColumnName 28
- TSSDatapoolFetch 29
- TSSDatapoolOpen 30

TSSDatapoolRewind 33
TSSDatapoolRowCount 34
TSSDatapoolSearch 35
TSSDatapoolSeek 36
TSSDatapoolValue 37
TSSDelay 61
TSSDisplay 79
TSSEnvironmentOp 48
TSSePrint 75
TSSErrorDetail 62
TSSGetComputerConfigurationAttributeList 63
TSSGetComputerConfigurationAttributeValue 64
TSSGetPath 65
TSSGetScriptOption 66
TSSGetTestCaseConfiguration 69
TSSGetTestCaseConfigurationAttribute 67
TSSGetTestCaseConfigurationAttributeList 68
TSSGetTestCaseName 70
TSSGetTestToolOption 70
TSSGetTime 50
TSSInternalvarGet 51
TSSInternalvarSet 101
TSSJavaApplicationStart 71
TSSLogCommand 102
TSSLogEvent 39
TSSLogMessage 41
TSSLogTestCaseResult 42
TSSNegExp 72
TSSPositionGet 79
TSSPositionSet 81
TSSPrint 76
TSSRand 73
TSSReportCommandStatus 82
TSSRunStateGet 83
TSSRunStateSet 83
TSSSeedRand 74
TSSServerStart 98
TSSServerStop 98, 99
TSSSharedVarAssign 87
TSSSharedVarEval 89
TSSSharedVarWait 90
TSSShutdown 100
TSSSyncPoint 93
TSSThink 53
TSSThinkTime 104
TSSTimerStart 54
TSSTimerStop 55
TSSUniform 76
TSSUniqueString 77
TTAddToSourceControl 122
TTCheckIn 124
TTCheckOut 125
TTClearFilter 127
TTConnect 128
 example 129
TTDisconnect 132
 example 132
TTEdit 133
TTExecuteNodeAction 135
TTExecuteSourceAction 136
TTGetChildren 138
TTGetFilterEx 142
TTGetIcon 138, 144
 example 144
TTGetIsFunctionSupported 145
TTGetName 147
 example 148
TTGetNode 149
TTGetNodeActions 151
TTGetRoots 152
TTGetSourceActions 153
TTGetSourceControlStatus 155
TTGetSourceIcon 157
TTGetTestToolOptions 158
TTGetTypeIcon 161
TTNew 162
 example 163
TTSelect 164
 example 166
TTSetConfiguration 166
TTSetFilterEx 168
TTShowProperties 170
 example 171
TTUndoCheckout 172
type node structure, using 195

U

- update, shared variable 87
- using the type node structure 195
- utility
 - list of functions 57

V

- Visual Basic test script type 2
- VU test script type 2

W

- wait
 - for application termination id 60