Rational TestManager Extensibility Reference

VERSION 2001A.04.00

PART NUMBER 800-024528-000

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

Rational®
the e-development company™

Contents

Part 2: Adding Custom Test Input Types

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*

  This part 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*

  This part 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 will 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 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
Built-in Test Script Types **on page 1**	Explains the role of test script types in execution and describes built-in test script types.
Adding Test Script Types **on page 2**	Describes the two ways to extend TestManager to support a new test script type or testing environment.
Adding a Command Line Test Script Type **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.
Adding a Custom Test Script Type **on page 7**	Describes the tasks required to create a custom test script type.
Component Description and Communication Overview **on page 8**	Lists test script adapter components and presents a diagram describing their interaction during playback.
Using the Command-line Execution Engine **on page 10**	Explains how to use `rttsee` to execute test scripts from a command line.

Built-in Test Script Types

When a Rational 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 SQABasic, 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 an application that runs in IBM's WebSphere Enterprise Java Beans (EJB) environment.
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 advantage of this method, illustrated in *Adding a Command Line Test Script Type* on page 3, 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.
- Unless such test scripts make explicit calls to the C Test Script Services (TSS) documented in Chapter 3, they will not be integrated into the TestManager logging, monitoring, and reporting framework.

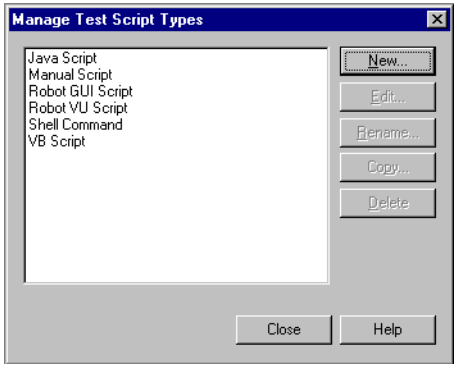
The other way to extend TestManager is to create a custom test script type. This method, described in *Adding a Custom Test Script Type* on page 7, requires that you develop programs implementing the C APIs described in chapters 2–4 of this manual. For example, you can create adapters for currently unsupported 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

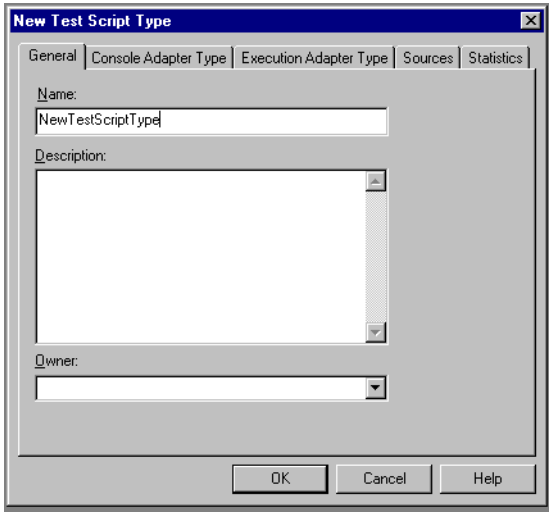
This example illustrates how, without doing any programming, to extend TestManager so that it can support test scripts written in a new source language. After you have followed these procedures, TestManager will be able to manage test scripts written in Perl. Specifically, a TestManager user will be able to 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.

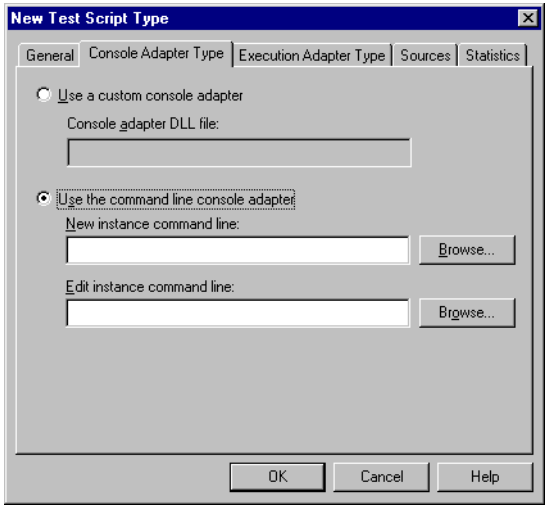


3 Click the **New** button. The New Test Script Type dialog box appears with the **General** tab selected.



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
```

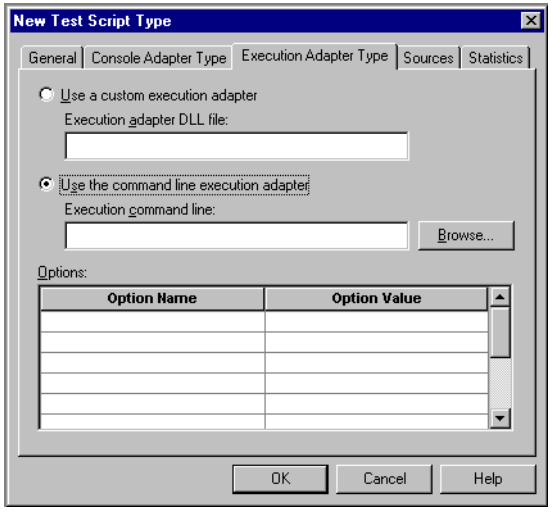
- In the **Edit instance command line** box, type the command to start in order to view or edit existing scripts of this type.For example:

  ```
  notepad {testscriptpath}
  ```

  Type `{testscriptpath}` exactly as shown.

The program you enter (in this case `wordpad`) 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:

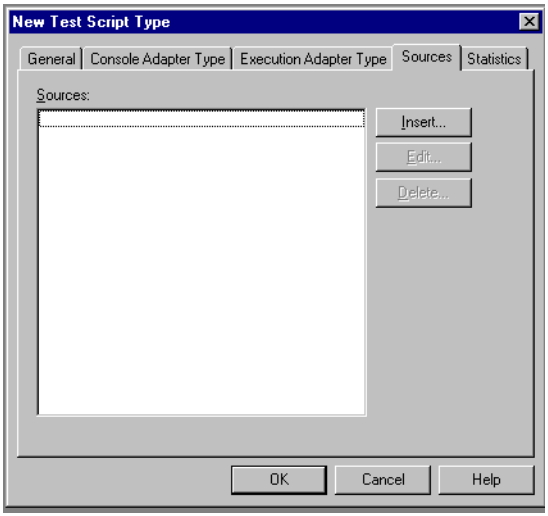
```
perl {testscriptpath}
```

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.)

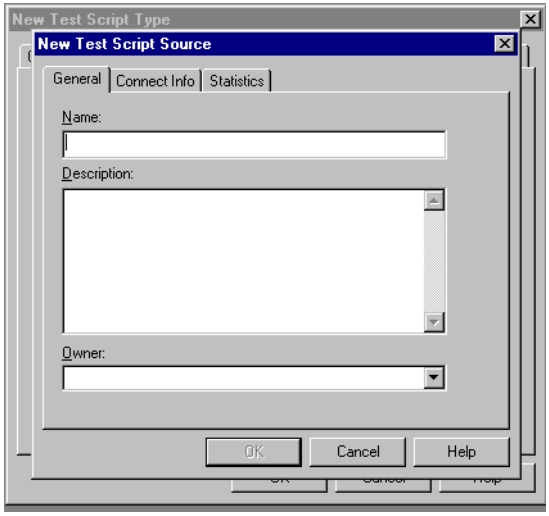
In the **Options** area, type the following Option Name and Option Value pair:

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

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



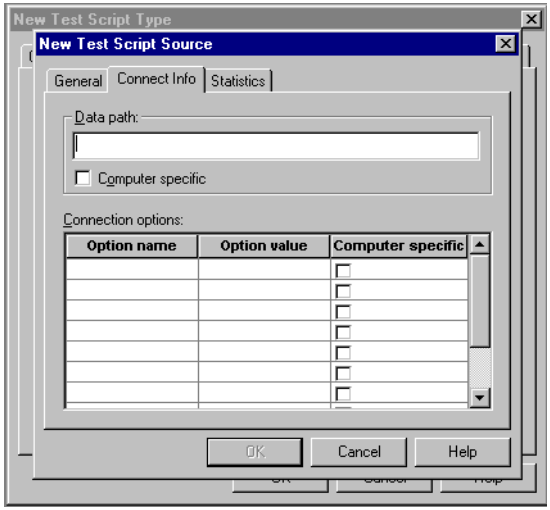
7 Click the **Insert** button. A popup appears telling you that the test script you are defining must be created before proceeding — answer **Yes**. The dialog box changes as shown below.



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.

8 Click the **Connect Info** 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 that will be used to edit or view scripts of the new type. These procedures are explained in the TestManager online Help and in the *Using Rational TestManager* manual.The procedures are similar to those described in *Adding a Command Line Test Script Type* on page 3.

Your code must reside in dynamically-linked 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.

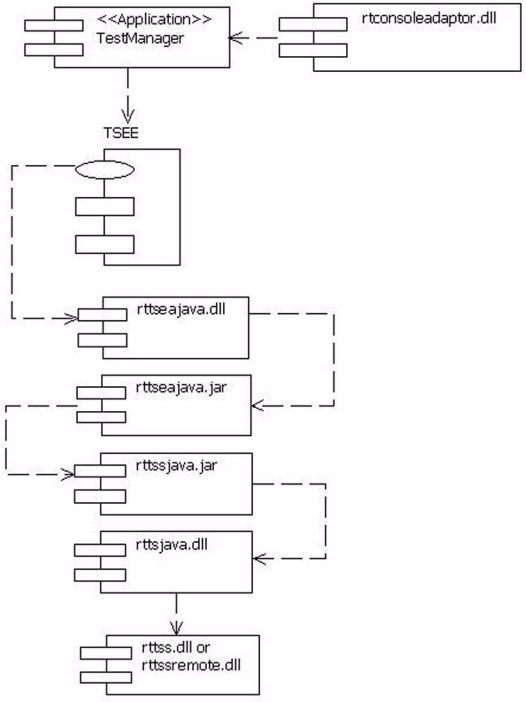
Component Description and Communication Overview

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
rttss.dll	Rational Test\	The C Test Script Services library. Used for scripts executed inside TSEE process space.
rttssremote.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.

File	Path in installation directory	Description
tsea	Rational Test\	Directory where TSEA components must reside.
rttss.lib	Rational Test\rtsdk\c\lib\	The TSS LIB file for linking.
rttssremote.lib	Rational Test\rtsdk\c\lib\	The TSS proxy LIB file for linking.
rttss.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 programs(s) used to edit or view the scripts. The TSCA used for Java test script types is rtconsoleadapter.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 will differ 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 dynamically-linked C library. This is the layer where requested services are performed and integrated into the TestManager UI. The Java TSEA supports direct script execution only and uses rttss.dll. A TSEA that supports proxy script execution uses rttssremote.dll.

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

Test Script Execution Adapter API

2

About This API

This chapter describes the Rational Test Script Execution Adapter (TSEA) API. This API defines the C language calls that Rational TestManager's 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

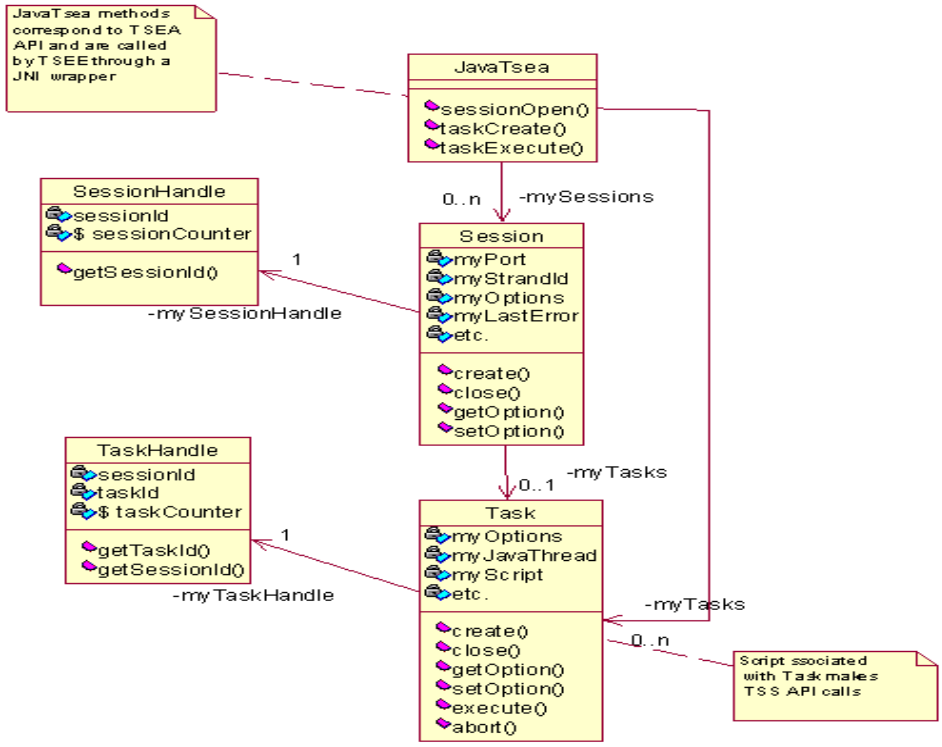
Communication between the TSEE and the TSEA occurs in three phases as described in the following steps.

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 Rational 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 rttss.h, these are the types of data that your TSEA receives from and returns to the TSEE.

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

Summary

The TSEA API includes the following calls.

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

SessionClose()

Closes a TSEA session.

Syntax

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

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

See Also

`SessionSetOption()`

SessionOpen()

Opens a session with a TSEA.

Syntax

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

Element	Description
hostname	The name (or IP adddres in dot notation) of the TSEA host.
port	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 `TSSConnect()` to start a proxy script execution process, this *port* must be passed to the process.
strandID	Strand (thread) ID for TSS calls in this session.
message	A statement that, if the open fails, will be 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 will be 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
session	The session handle, returned by SessionOpen().
optname	The session option whose value is to be set.
optval	The new value of *optname*.
len	The size of buffer *optval*.

See Also

SessionGetOption()

TaskAbort()

Aborts a TSEA task.

Syntax

s32 **TaskAbort** (TaskHandle *task*)

Element	Description
task	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
task	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

```
s32 TaskCreate (SessionHandle session, TaskType type, char
   *sourcelocation, char *taskFile)
```

Element	Description
session	The session handle, returned by `SessionOpen()`.
type	The test script type for scripts that this TSEA plays back.
sourcelocation	The location where source scripts of *type* are located.
taskFile	The name of the file containing the test script.

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
task	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 otherwise.

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

See Also

`TaskSetOption()`

TaskSetOption()

Sets the value of a task option.

Syntax

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

Element	Description
task	The task handle, returned by `SessionOpen()`.
optname	The task option whose value is to be set.
optval	The new value of *optname*.
len	The size of buffer *optval*.

See Also

`TaskGetOption()`

TSEAError()

Gets a message following an error.

Syntax

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

Element	Description
session	The session handle, returned by SessionOpen().
message	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.

Test Script Services 3

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 will be 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 multi-computer 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 *Using Rational TestManager* manual. 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
`TSSDatapoolClose()`	Closes a datapool.
`TSSDatapoolColumnCount()`	Returns the number of columns in a datapool.
`TSSDatapoolColumnName()`	Returns the name of the specified datapool column.
`TSSDatapoolFetch()`	Moves the datapool cursor to the next row.
`TSSDatapoolOpen()`	Opens the named datapool and sets the row access order.
`TSSDatapoolRewind()`	Resets the datapool cursor to the beginning of the datapool access order.
`TSSDatapoolRowCount()`	Returns the number of rows in a datapool.
`TSSDatapoolSearch()`	Searches a datapool for the named column with a specified value.
`TSSDatapoolSeek()`	Moves the datapool cursor forward.

Function	Description
TSSDatapoolValue()	Retrieves the value of the specified datapool column.

TSSDatapoolClose()

Closes a datapool.

Syntax–

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

Element	Description
dpid	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_NOSERVER. 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
dpid	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
dpid	The ID of the datapool. Returned by `TSSDatapoolOpen()`.

Element	Description
columnNumber	A positive number indicating the number of the column whose name you want to retrieve. The first column is number 1.

Return Value

On success, this 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
dpid	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.

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 curson 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
name	The name of the datapool to open. If *accessFlags* includes TSS_DP_NO_OPEN, no CSV datapool is opened; instead, *name* will refer to the contents of *overrides* specifying a one-row table. Otherwise, the CSV file *name* in the Rational project is opened.
accessFlags	Optional flags indicating how the datapool is accessed when a script is played back. Specify at most one value from each of the following categories: **1** Specify the sequence in which datapool rows are accessed: 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: 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: TSS_DP_SHARED – all virtual testers work from the same access order (default) TSS_DP_PRIVATE – virtual testers each work from their own sequential, random, or shuffle 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 will be numbered 101. Not valid with TSS_DP_RANDOM or TSS_DP_PRIVATE. **5** TSS_DP_REWIND specifies that the datapool should be rewound when opened. Can be used only with TSS_DP_PRIVATE. **6** TSS_DP_NO_OPEN specifies that, instead of a CSV file, the opened datapool will consist only of column/value pairs specified in a local array *overrides*[].
overrideCount	The number of columns in array *overrides*. Must be greater than 0 if access flag TSS_DP_NO_OPEN is specified; otherwise, must be 0.
overrides	A local, two-dimensional array of column/value pairs, where *overrides*[n].*name* is the column name and *overrides*[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_NOSERVER`. No previous successful call to `TSSConnect()`.
- `TSS_INVALID`. The *`accessFlags`* are 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 *`accessFlags`* are specified as 0 or, the rows are accessed in the default order: sequentially, with no wrapping, and with a shared 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
dpid	The ID of the datapool. Returned by `TSSDatapoolOpen()`.

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 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.

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, 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
dpid	The ID of the datapool. Returned by `TSSDatapoolOpen()`.

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_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 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
dpid	The ID of the datapool. Returned by `TSSDatapoolOpen()`.
keycount	The number of columns in *keys*.
keys	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
dpid	The ID of the datapool. Returned by TSSDatapoolOpen().
count	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
`dpid`	The ID of the datapool. Returned by `TSSDatapoolOpen()`.
`columnName`	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_NOSERVER`. 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 will be 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 will come 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
eventType	Contains the description to be displayed in the log for this event.

Element	Description
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ 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.
description	Contains the string to be put in the entry's failure description field.
propertyCount	Specifies the number of rows in the property array.
property	An array containing property name/value pairs, where property[n].name is the property name and property[n].value 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_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_levelandEVAR_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
message	Specifies the string to log.
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ `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.
description	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. 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
testcase	Identifies the test case whose result is to be logged.
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ 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.
description	Contains the string to be displayed in the event of a log failure.
propertyCount	Specifies the number of rows in the property array.
property	An array containing property name/value pairs, where property[n].name is the property name and property[n].value 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.

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 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 testcase 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
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ 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.
description	Contains the string to be displayed in the event of failure.
starttime	An integer indicating a timestamp to override the timestamp set by TSSCommandStart(). To use the timestamp set by TSSCommandStart(), specify as 0.
endtime	An integer indicating a timestamp to override the current time. To use the current time, specify as 0.
logdata	Text to be logged describing the ended command.
propertyCount	Specifies the number of rows in the property array.
property	An array containing property name/value pairs, where property[n].name is the property name and property[n].value 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.

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
label	The name of the timer to be started and logged, or NULL for an unlabeled timer.
name	The name of the command to time.
state	The run state to log with the timed command. See the run state table starting on page 73.

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

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 73.

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

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
envVar	The environment variable to operate on. Valid values are listed in the `EvarKey` enumeration.
envOP	The operation to perform. Valid values are listed in the `EvarOp` enumeration.
envVal	The value operated on as specified by *envOP* to produce the new value for *envVar*.

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

Environment variables define and control the enviromnent 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
};
```

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_NOSERVER`. 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
`internVar`	The internal variable to operate on. Defined in the `IVKey` enum structure shown on page 50.

Element	Description
ivVal	OUTPUT. The returned value of the specified *internVar*.

Return Value

This function returns one of the following values:

- `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

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_ncnull,
   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)
```

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

Return Value

This 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

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 46.

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
label	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.
timeStamp	An integer specifying a timestamp 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 timestamp 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
label	The name of the timer to be stopped and logged, or NULL for an unlabeled timer.
timeStamp	If specified as 0, the current time is recorded.
rmFlag	Specify as 0 to stop the timer without removing it; otherwise, specify as non-zero. 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 timestamp 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
`TSSDelay()`	Delays the specified number of milliseconds.
`TSSErrorDetail()`	Retrieves error information about a failure.
`TSSGetScriptOption()`	Gets the value of a script playback option.
`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.
`TSSNegExp()`	Gets the next negative exponentially distributed random number with the specified mean.
`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.

TSSDelay()

Delays script execution for the specified number of milliseconds.

Syntax

```
s32 TSSDelay(s32 msecs)
```

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

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

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.

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

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_NOSERVER`. 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);
}
```

TSSGetScriptOption()

Gets the value of a script playback option.

Syntax

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

Element	Description
`optionName`	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_NOSERVER`. 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 13.

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()`*

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_NOSERVER`. 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_NOSERVER. 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 TSSTestCaseName() 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");
```

TSSNegExp()

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

Syntax

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

Element	Description
mean	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_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.

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.

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
seed	The base integer.

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

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
message	The string to print.

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.

Example

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

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

See Also

`TSSPrint()`

TSSPrint()

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

Syntax

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

Element	Description
message	The string to print.

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.

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
low	The low end of the range.
high	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_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.

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 distriburted random number between –10 and 10.

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

See Also

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

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 TestManager's 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
message	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_NOSERVER`. No previous successful call to `TSSConnect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

Comments

This message will be 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
srcFile	OUTPUT. The name of a source file. After a successful call, this variable will contain the name of the source file that was specified with the most recent `TSSPositionSet()` call.
lineNumber	OUTPUT. The name of a local variable. After a successful call, this variable will contain the current line position in *srcFile*.

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_NOSERVER. 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 will be 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 that will 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
srcFile	The name of the test script, or NULL for the current test script.

Element	Description
lineNumber	The number of the line in *srcFile* 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_NOSERVER. 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 will be 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
status	The status of a command. Can be one of the following: ▪ 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_NOSERVER. 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 73. 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

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

Return Value

This function exits with one of the following results:

- TSS_OK. Success.
- TSS_NOSERVER. 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 SQABasic 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 on http connection
MST_HTTPDISC	waiting on http disconnect
MST_IIOP_INVOKE	iiop_invoke in progress
MST_INCL	mask including above basic states
MST_INITTASK	initializing task
MST_ITDLY	inter-task 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 on socket connection
MST_SOCKDISC	waiting on socket disconnect
MST_SQABASIC_CODE	running SQABasic code
MST_SQLCONN	waiting on SQL client connection
MST_SQLDISC	waiting on SQL client disconnect
MST_SQLEXEC	executing SQL statements
MST_STARTAPP	SQABasic: starting app
MST_SUSPENDED	suspended
MST_TEST	test case, emulate
MST_THINK	thinking

Run State	Meaning
MST_TRN_PACING	transactor pacing delay
MST_TUXEDO	Tuxedo execution
MST_TYPE	typing
MST_USERCODE	SQAVu user code
MST_INIT	doing start-up initialization
MST_UNDEF	user's micro_state is undefined
MST_WAITOBJ	SQABasic: waiting for object
MST_WAITRESP	waiting for response
MST_WATCH	interactive -W watch record
MST_XCLNTCONN	waiting on http connection
MST_XCLNTCONN	waiting on socket connection
MST_XCLNTCONN	waiting on SQL client connection
MST_XCLNTCONN	waiting on X client connection
MST_XCLNTDISC	waiting on http disconnect
MST_XCLNTDISC	waiting on socket disconnect
MST_XCLNTDISC	waiting on SQL client disconnect
MST_XCLNTDISC	waiting on 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 sychronize 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
name	The name of the shared variable to operate on.
value	The right-hand-side value of the assignment expression.

Element	Description
op	Assignment operator. Can be one of the following: ▪ SHVOP_assign ▪ SHVOP_add ▪ SHVOP_subtract ▪ SHVOP_multiply ▪ SHVOP_divide ▪ SHVOP_modulo ▪ SHVOP_and ▪ SHVOP_or ▪ SHVOP_xor ▪ SHVOP_shiftleft ▪ SHVOP_shiftright
returnVal	OUTPUT. If not specified as NULL, the resulting value of *name* after application of *op 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("foo",5,SHVOP_add,NULL)` is equivalent to `foo += 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
name	The name of the shared variable to operate on.
value	OUTPUT. A local container into which the value of *name* is retrieved.
op	Increment/decrement operator for the returned value: Can be one of the following: ▪ SHVADJ_none SHVADJ_pre_inc ▪ SHVADJ_post_inc ▪ SHVADJ_pre_dec ▪ SHVADJ_post_dec

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_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 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_inc);
```

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
name	The name of the shared variable to operate on.
min	The low range for the value of *name*.
max	The high range for the value of *name*.

Element	Description
adjust	The value to increment/decrement the named shared variable by once it meets the *min* – *max* range.
timeout	The timeout preference (how long to wait for the condition to be met). Enter one of the following: ▪ A negative number for no timeout. ▪ 0 to return immediately with an exit value of 1 (condition met) or 0 (not met) ▪ The number of milliseconds to wait for the value of *name* to meet the criteria, before timing out with and returning an exit value of 1 (met) or 0 (not met).
returnVal	OUTPUT. The value of *name* at the time of the return, before any possible adjustment. If *timeout* expired before the return, the value is not adjusted. Otherwise, *returnVal* is incremented/decremented by *adjust*.

Return Value

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

- 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

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 utilizing 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 multi-user 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 multi-user 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, then 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
label	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_NOSERVER. No previous successful call to TSSConnect().
- TSS_INVALID. The synchronication 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, you will want to insert synchronization points into a TestManager suite rather than inserting the TSSSyncPoint() call into 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 TSSSyncPoint(), 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 = TSSSyncPoint("BlockUntilSaveComplete");
```

Session Services

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 will be started and each will remain active until all scripts of the respective types finish.

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

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

Summary

Use the session functions listed in the following table to manage proxy TSS servers and sessions. These functions are not needed for sessions that are directly executed by TestManager.

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

TSSConnect()

Connects to a TSS proxy server.

Syntax

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

Element	Description
host	The name (or IP address in quad dot notation) of the host on which the proxy TSS server process is running.
port	The listening port for the TSS server on *host*, or `0` (recommended) to let TestManager select the port.
id	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, then *id* is ignored. (You cannot change *id*.)

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
ctx	The type of context information to pass: Can be one of the following: ▪ CTXT_workingDir ▪ CTXT_datapoolDir ▪ CTXT_timeZero ▪ CTXT_todZero ▪ CTXT_logDir ▪ CTXT_logFile ▪ CTXT_logData ▪ CTXT_testScript ▪ CTXT_style ▪ CTXT_sourceUID
value	The information of type *ctx* 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 stand-alone — 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 stand-alone and also by a proxy TSS server 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()

Disconnects from a TSS proxy server.

Syntax

```
void TSSDisconnect (void)
```

Return Value

None.

Comments

This call closes the connection established by `TSSCconnect()` 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
port	The listening port for the TSS server. If specified as 0 (recommended), the system chooses the port and returns its number to *port*.

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

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_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 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
internVar	The internal variable to operate on. Internal variables and their values are listed in the table startingon page 50. See page 50 for the IVKey and page 51 for the IVValue definitions.
ivVal	The new value for *internVar*.

Return Value

The 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

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
name	The command name.
label	The event label.
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ 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.
description	Contains the string to be displayed in the event of failure.
starttime	An integer indicating a timestamp. If specified as 0, the logged timestamp will be the later of the values contained in internal variables IV_fcs_ts and IV_fcr_ts.
endtime	An integer indicating a timestamp. If specified as 0, the time set by TSSCommandEndis logged.
logdata	Text to be logged describing the ended command.
propertyCount	Specifies the number of rows in the property array.
property	An array containing property name/value pairs, where property[n].name is the property name and property[n].value 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.

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

Return Value

On success, this 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()`

Test Script Console Adapter API

4

This chapter provides reference information on how to build a custom Test Script Console Adapter (TSCA).

About the Test Script Console Adapter

A Test Script Console Adapter (TSCA) is a dynamic-link library (DLL) that manages one or more test script types. TestManager supports two categories of Test Script Console Adapters:

- Command-line TSCA. This TSCA is for file-based test scripts that have operations executed from a command line — for example, PERL scripts. File based means that the individual test scripts can be accessed using the standard Microsoft File Open dialog box.

  Test Manager has provided a TSCA that accommodates any file-based test script type driven from the command line. Using this console adapter requires no programming; you only need to specify the executable commands for creating and editing a script. For more information about using TestManager's built-in console adapter, see the TestManager online Help.

- Custom TSCA. This TSCA can be used for file-based scripts, but must be used for test script types that are not file based — for example, Rational ManualTest scripts. For these kinds of scripts, you must implement your own DLL that provides at least a minimal user interface and enables TestManager to connect to and disconnect from the test input source.

  TestManager provides an applications programming interface (API) for implementing a custom TSCA. This API enables you to provide a user interface for the tester to create, edit, and select scripts of the type. You do not have to implement all of the functions available in the API. The only required functions are `TTConnect()`, `TTDisconnect()`, and `TTSelect()`. If your TSCA omits any other function, the operations defined in that function are not available.

Summary of TSCA Functions

The TSCA API includes the following functions:

Function	Description
`TTConnect()`	Establishes a connection to a test source.
`TTDisconnect()`	Disconnects from the test source.
`TTEdit()`	Displays a test script in an appropriate editor.
`TTGetIcon()`	Returns the path of the icon that identifies scripts of this custom type.
`TTGetName()`	Returns the name of the specified test script.
`TTNew()`	Enables the tester to create a new script for this test type using the hosted tool.
`TTSelect()`	Enables the tester to select a test script.
`TTShowProperties()`	Displays the properties of a test script.

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

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

TTConnect()

Establishes a connection to the source of test inputs.

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
ConnectInfo	INPUT. A string that specifies the location of the test source, typically a path.
UserID	INPUT. A string that identifies the current user.
pConnectOptions	INPUT. A pointer to a list of structures that specify the name-value pair of connection options.
nOptions	INPUT. An integer that specifies the total number of connection options.
SourceID	OUTPUT. A string that uniquely identifies the test source. The ID is used in subsequent calls to the adapter.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_ERROR_UNABLE_TO_CONNECT. No connection with the test source was possible.
- TTYPE_ERROR_INVALID_CONNECTINFO. The adapter was unable to use the connection information.
- TTYPE_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[TTYPE_MAX_NAME];
char Value[TTYPE_MAX_NAME];
}ConnectOptionType;
```

The connection information is specified by an administrative user in the TestManager New Test Script Source dialog box and then passed into this function in *`ConnectInfo`*. After the connection has been established, you assign a unique identifier for the test source to *`SourceID`*. TestManager uses this identifier for subsequent calls to the adapter to identify a particular connection. Be sure to document the format of this string. This is important if your adapter supports multiple, simultaneous connections.

You can use the string *`ErrorDescription`* to provide the tester with additional information.

Example

```
//*********************************************************************
// TTConnect
//
//   Remarks:
//
//      This function creates a connection to a data store of test
//      scripts using the provided connection information.  It returns
//      a handle that the client uses for subsequent calls to the
//      adapter.
//
//   Parameters:
//
//      const CHAR ConnectInfo[TTYPE_MAX_PATH] [input]
//         Contains information that identifies the datastore of test
//         scripts. Often, this parameter is a path.
//
//      const CHAR UserID[TTYPE_MAX_ID] [input]
//         The TestManager User ID of the user connecting to
//         the data store.
//
//      ConnectOption  *pConnectOptions[] [input]
//         A user-defined array of test script source options defined
//         in TestManager.
//
//      int nOptions [input]
//         The number of connection options.
//
//      CHAR SourceID[TTYPE_MAX_ID] [output]
//         The handle that the client uses to identify the connection
```

```
//         to the data store in subsequent calls to the adapter.
//
//      CHAR ErrorDescription[TTYPE_MAX_ERROR] [output]
///        An error description returned by the server if the
//         connect method fails.
//
//   Return Value:
//
//      TTYPE_SUCCESS
//
//      TTYPE_ERROR_UNABLE_TO_CONNECT
//         The connection failed for some unknown reason.
//
//   Notes:
//
//      In this example adapter, the path of the data store is used as
//      the key to our connection map. The connection map associates
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the data store.
//
//*********************************************************************
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])
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());

   HRESULT rc = TTYPE_SUCCESS;
   CString sType = "";

   CString sConnectionIdentifier = ConnectInfo;

   // The identifier that is passed back and subsequently is
   // passed back into the adapter on almost every call is the
   // ConnectInfo (typically a path to the data store).
   _tcscpy(SourceID, (LPCTSTR) sConnectionIdentifier);

   CConnectionContext *pExistingContext=0;

   // See if there is an existing connection to this source by looking
   // up the context information for the specified
   // ConnectionIdentifier.
   m_ServerConnections.Lookup(sConnectionIdentifier, (void
   *&)pExistingContext);

   // If the context was found, then return.
   if (pExistingContext)
      return S_OK;

   // Not connected yet.
   try
   {
```

```
   CString sUserName, sPassword;

// For this example, assume that connection options include the
// UserName and Password necessary to connect to the datastore.

   // Retreive 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);

   }
   catch (_com_error)
   {
      rc = TTYPE_ERROR_UNABLE_TO_CONNECT;
   }
   return rc;
```

See Also

`TTDisconnect()`

TTDisconnect()

Disconnects from the test source.

Syntax

```
HRESULT TIDisconnect(const char SourceID[TTYPE_MAX_ID], char
   ErrorDescription[TTYPE_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that TestManager uses to identify the connection.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns one of the following values:

- TTYPE_SUCCESS. The function completed successfully.
- TTYPE_ERROR_INVALID_SOURCE_ID. The specified source information was not correct.
- TTYPE_ERROR_UNABLE_TO_DISCONNECT. There was no existing connection to disconnect from.
- TTYPE_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 input source.

You can use the string *ErrorDescription* to provide the tester with additional information.

Example

```
//*********************************************************************
// TTDisconnect
//
//    Remarks:
//
//       This function disconnects from an active datastore connection.
//
//    Parameters:
//
//       const CHAR SourceID[TTYPE_MAX_ID] [input]
//          The handle that the client uses to identify the connection
//          to the datastore.
//
```

```
//      CHAR ErrorDescription[TTYPE_MAX_ERROR] [output]
//         An error description returned if the disconnect method
//         fails.
//
//   Return Value:
//
//      TTYPE_SUCCESS
//
//      TTYPE_ERROR_INVALID_SOURCEID
//         The specified SourceID did not map to an active connection.
//
//      TTYPE_ERROR_UNABLE_TO_DISCONNECT
//         The disconnect failed.
//
//   Notes:
//
//      In this example adapter, the path of the data store is used as
//      the key to our connection map.  The connection map associates
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the data store.
//
//*************************************************************************
HRESULT TTDisconnect(const CHAR SourceID[TTYPE_MAX_ID],  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, 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()

Displays a test script in the appropriate editor.

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
SourceID	INPUT. A string that identifies the connection.
ScriptID	INPUT. Astring that identifies the test script.
LineNumber	INPUT. An integer that specifies where the cursor is placed when the file is opened for editing.
pScriptOptions	INPUT and OUTPUT. The script options passed from `TTSelect()`. Any changes made to the script options are passed back and made available to the test case or suite.
nScriptOptions	INPUT and OUTPUT. An integer that specifies the number of script options.
ErrorDescription	OUTPUT. A string for customized error messages.
lWindowContext	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_SOURCE_ID`. 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 can sometimes be important for the script to execute properly. The TSCA can specifiy 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[TTYPE_MAX_NAME];
char Value[TTYPE_MAX_NAME];
}ScriptOptionType;
```

You can use the string *ErrorDescription* to provide the user with additional information.

Example

```
//*********************************************************************
// TTEdit
//
//    Remarks:
//
//       This function launches the editor for a specific test script.
//
//    Parameters:
//
//       const CHAR SourceID[TTYPE_MAX_ID] [input]
//          The handle that the client uses to identify the connection
//          to the datastore.
//
//       const CHAR ScriptID[TTYPE_MAX_ID] [input]
//          The unique ID that identifies a specific test script in the
//          datastore.
//
//       CHAR ErrorDescription[TTYPE_MAX_ERROR] [output]
//          An error description returned by the adapater if the test
//          script cannot be accessed.
//
//       long lWindowContext [input]
//          A handle to a window, which can be the parent of a dialog
//          or property sheet displayed by this method.
//
//    Return Value:
//
//       TTYPE_SUCCESS
//
//       TTYPE_ERROR_INVALID_SOURCEID
//
//    Notes:
```

```
//
//      In this example adapter, the path of the data store is used as
//      the key to our connection map.  The connection map associates
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the data store.
//
//*********************************************************************
HRESULT TTEdit(const CHAR SourceID[TTYPE_MAX_ID],const CHAR
ScriptID[TTYPE_MAX_ID], ScriptOption *pScriptOptions[], int
*nScriptOptions, CHAR ErrorDescription[TTYPE_MAX_ERROR], long
lWindowContext)
{
   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: 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(), TTProperties(), TTSelect()

TTGetIcon()

Returns the path of the icon that identifies scripts of this test type.

Syntax

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

Element	Description
SourceID	INPUT. A string that identifies the connection.
IconPath	OUTPUT. Astring that specifies the path where the image file of the icon is located. The file must contain a 16 x 16 bitmap image.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns one of the following values:

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

Comments

You can use the string *ErrorDescription* to provide the user with additional information.

Example

```
//*********************************************************************
// TTGetIcon
//
//    Remarks:
//
//       This function returns the path to the icon that represents
//       the test script type.
//
//    Parameters:
```

```
//
//      const CHAR SourceID[TTYPE_MAX_ID] [input]
//         The handle that the client uses to identify the connection
//         to the datastore.
//
//      CHAR  IconPath[TTYPE_MAX_NAME] [output]
//         The path of the selected type's icon
//
//      CHAR ErrorDescription[TTYPE_MAX_ERROR] [output]
//         An error description returned by the adapter if it cannot
//         obtain the icon.
//
//   Return Value:
//
//      TTYPE_SUCCESS
//
//      TTYPE_ERROR_INVALID_SOURCEID
//
//   Notes:
//
//      In this example adapter, the path of the data store is used as
//      the key to our connection map.  The connection map associates
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the data store.
//
//*********************************************************************
HRESULT TTGetIcon(const CHAR SourceID[TTYPE_MAX_ID], CHAR
IconPath[TTYPE_MAX_PATH], CHAR ErrorDescription[TTYPE_MAX_ERROR])
{
   DWORDdwError;
   charszModuleFileName[_MAX_PATH+1];
   charszModuleFilePath[_MAX_PATH+1];
   charszDir[_MAX_DIR+1];
   charszDrive[_MAX_DRIVE+1];

   CConnectionContext *pContext=0;
   CString sSourceID = SourceID;

   // Lookup the ConnectionContext based on the specified SourceID.
   m_ServerConnections.Lookup(sSourceID, (void *&)pContext);

   // If the context was found, continue.
   if (pContext)
   {
      // In this example, the bitmap files are co-located with the
      // adapter. Get the location of the adapter.
      dwError=GetModuleFileName((HMODULE)AfxGetInstanceHandle(),
      szModuleFileName, _MAX_PATH);
      _splitpath(szModuleFileName, szDrive, szDir, NULL, NULL);

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

```
        // In this example, the existence of a file name
        // guiscript.bmp is assumed.
        // In your adapter, you reference the appropriate file
        _tcscat(szModuleFilePath, "guiscript.bmp");

        // Return the path.
        _tcscpy(IconPath, szModuleFilePath);
    }
    return TTYPE_SUCCESS;
}
```

See Also

`TTEdit()`, `TTProperties()`

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
SourceID	INPUT. A string that identifies the connection.
ScriptID	INPUT. A string that identifies the test script.
ScriptName	OUTPUT. A string that specifies the name of a test script.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns on of the following values:

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

Comments

You can use the string *ErrorDescription* to provide the user with additional information.

Example

```
//*********************************************************************
// TTGetName
//
//    Remarks:
//
//       This function returns the name of a test script.  In the
//       Console adapter for Rational test script types, the script
//       ID is also the script name.  However, this does not necessarily
//       need to be the case.  The ID could be some internal key,
//       while the name is displayable.
//
//
//    Parameters:
//
//       const CHAR SourceID[TTYPE_MAX_ID] [input]
//          The handle that the client uses to identify the connection
//          to the datastore.
//
//       const CHAR ScriptID[TTYPE_MAX_ID] [input]
//          The unique identifier of the test script whose name is
//          retreived.
//
//       CHAR  ScriptName[TTYPE_MAX_NAME] [output]
//          The returned test script name.
//
//       CHAR ErrorDescription[TTYPE_MAX_ERROR] [output]
//          An error description if the adapter cannot obtain
//          the name of the specified test script.
//
//    Return Value:
//
//       TTYPE_SUCCESS
//
//       TTYPE_ERROR_INVALID_SOURCEID
//
//    Notes:
//
//       In this example adapter, the path of the data store is used as
//       the key to our connection map.  The connection map associates
//       an instance of CConnectionContext with each active connection.
//       Each instance of CConnectionContext stores critical
//       information about the connection to the data store.
//
//*********************************************************************
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.*/

   _tcscpy(ScriptName, sScriptName);

   return rc;
}
```

See Also

TTEdit(), TTProperties()

TTNew()

Enables the tester to create a new script for this type of test using the hosted tool.

Syntax

```
HRESULT TTNew(const char SourceID[TTYPE_MAX_ID], char
   ScriptID[TTYPE_MAX_ID], char Name[TTYPE_MAX_NAME],
   ScriptOption *pScriptOptions[], int* nScriptOptions, char
   ErrorDescription[TTYPE_MAX_ERROR], long lWindowContext)
```

Element	Description
SourceID	INPUT. A string that identifies the test source.
ScriptID	OUTPUT. A string that identifies the test script.
Name	OUTPUT. A string that specifies the name of a test script.
pScriptOptions	OUTPUT. Reserved for future use.
nScriptOptions	OUTPUT. Reserved for future use.
ErrorDescription	OUTPUT. A string for customized error messages.
lWindowContext	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_SOURCE_ID`. The input 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 script, this function passes the ID and script name to Test Manager. TestManager passes this information to the `TTEdit()` and `TTProperties()` functions of this adapter as well as to the Test Script Execution Adapter (TSEA).

You can use the string *`ErrorDescription`* to provide the user with additional information.

Example

```
//**********************************************************************
// TTNew
//
//    Remarks:
//       This function enables the creation of a new test script.
//
//    Parameters:
//
//       const CHAR SourceID[TTYPE_MAX_ID] [input]
//          The handle that the client uses to identify the connection
//          to the datastore.
//
//       CHAR ScriptID[TTYPE_MAX_ID] [input]
//          The unique ID that identifies a specific test script in the
//          datastore.
//
//       CHAR ErrorDescription[TTYPE_MAX_ERROR] [output]
//          An error description returned by the adapter if the test
//          script cannot be accessed.
//
//       long lWindowContext [input]
//          A handle to a window that can be the parent of a dialog
//          or property sheet displayed by this method.
//
//    Return Value:
//
//       TTYPE_ERROR
```

```
//
//      TTYPE_ERROR_INVALID_SOURCEID
//
//   Notes:
//
//      In this example adapter, the path of the data store is used as
//      the key to our connection map.  The connection map associates
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the data store.
//
//*********************************************************************
HRESULT TTNew(const CHAR SourceID[TTYPE_MAX_ID], CHAR ScriptID
[TTYPE_MAX_ID], CHAR Name[TTYPE_MAX_NAME], ScriptOption
*pScriptOptions[], int* nScriptOptions, CHAR
ErrorDescription[TTYPE_MAX_ERROR], long lWindowContext)
{
   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: 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(), TTProperties()

TTSelect()

Enables the tester to select a test script of this type.

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
SourceID	INPUT. A string that identifies the connection.
ScriptID	OUTPUT. Astring that identifies the test script.
ScriptName	OUTPUT. Astring that specifies the name of a test script.
pScriptOptions	OUTPUT. A structure that specifies options for running the test script.
nOptionOptions	OUTPUT. An integer that specifies the number of script options.
ErrorDescription	OUTPUT. A string for customized error messages.
lWindowContext	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_SOURCE_ID. The input 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 an interface for the user to select a test script of this type. The function returns the ID and name of the script to TestManager. These parameters are passed to the TTEdit() and TTProperties() functions and are also passed to the Test Script Execution Adapter (TSEA).

Script options are settings that may be important for the script to execute properly. The TSCA can specifiy 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;
```

You can use the string *`ErrorDescription`* to provide the user with additional information.

Example

```
//*********************************************************************
// TTSelect
//
//    Remarks:
//
//       This function displays a user interface (for example, a
//       dialog box), which enables a user to select a test script from
//       the datastore.
//
//    Parameters:
//
//       const CHAR SourceID[TTYPE_MAX_ID] [input]
//          The handle that the client uses to identify the connection
//          to the datastore.
//
//       CHAR ScriptID[TTYPE_MAX_ID] [output]
//          The unique ID that identifies the selected test script in
//          the datastore.
//
//       CHAR ScriptName[TTYPE_MAX_ID] [output]
//          The name of the test script.
//
//       CHAR ErrorDescription[TTYPE_MAX_ERROR] [output]
//          An error description returned by the adapter if the test
//          script cannot be accessed.
//
//       long lWindowContext [input]
//          A handle to a window displayed by this method.
//
//    Return Value:
//
//       TTYPE_ERROR
//
//       TTYPE_ERROR_INVALID_SOURCEID
//
//
//    Notes:
//
```

```
//      In this example adapter, the path of the data store is used as
//      the key to our connection map.  The connection map associates
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the data store.
//
//*********************************************************************
HRESULT TTSelect(const CHAR SourceID[TTYPE_MAX_ID], CHAR ScriptID
[TTYPE_MAX_ID], CHAR ScriptName[TTYPE_MAX_NAME], ScriptOption
*pScriptOptions[], int* nScriptOptions, CHAR
ErrorDescription[TTYPE_MAX_ERROR], long lWindowContext)
{
   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)
   {
   CString sScriptID, sScriptName;

   /* CODE OMITTED: Display the selection UI. If a test script is
      selected, copy its ID and Name in the variables sScriptName and
      sScriptID.*/

   _tcscpy(ScriptID, (const char *)sScriptID);
   _tcscpy(ScriptName, (const char *)sScriptName);

   }
   else
      rc = TTYPE_ERROR_INVALID_SOURCEID;

   return rc;
}
```

See Also

TTEdit(), TTProperties()

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
SourceID	INPUT. Astring that identifies the connection.
ScriptID	INPUT. A string that identifies the test script.
pScriptOptions	INPUT and OUTPUT.The script options passed in from `TTSelect()`. Any changes made to the script options must be passed back and made available to the test case or suite.
nScriptOptions	INPUT and OUTPUT. An integer that specifies the number of script options.
ErrorDescription	OUTPUT. A string for a customized error message.
lWindowContext	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_SOURCE_ID`. 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 script to execute properly. The TSCA can specifiy 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;
```

You can use the string *ErrorDescription* to provide the user with additional information.

Example

```
//*********************************************************************
// TTShowProperties
//
//    Remarks:
//
//       This function displays the properties of the test script
//       identified by the value of the ScriptID parameter.
//
//    Parameters:
//
//       const CHAR SourceID[TTYPE_MAX_ID] [input]
//          The handle that the client uses to identify the connection
//          to the datastore.
//
//       const CHAR ScriptID[TTYPE_MAX_ID] [input]
//          The unique ID that identies a specific test script in the
//          datastore.
//
//       CHAR ErrorDescription[TTYPE_MAX_ERROR] [output]
//          An error description returned by the adapter if the
//          properties cannot be displayed.
//
//       long lWindowContext [input]
//
//          A handle to a window or property sheet displayed by this
//          method.
//
//    Return Value:
//
//       TTYPE_SUCCESS
//
//       TTYPE_ERROR
//
//       TTYPE_ERROR_INVALID_SOURCEID
//
//    Notes:
//
//       In this example adapter, the path of the data store is used as
//       the key to our connection map.  The connection map associates
//       an instance of CConnectionContext with each active connection.
//       Each instance of CConnectionContext stores critical
//       information about the connection to the data store.
```

```
//
//*****************************************************************
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.*/

   }
   else
      rc = TTYPE_ERROR_INVALID_SOURCEID;

   return rc;
}
```

See Also

TTSelect(), TTEdit()

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.

Part 2: Adding Custom Test Input Types

Introduction to Extensible Test Inputs

5

This chapter provides an overview of process of implementing a Test Input Adapter (TIA) with a few general examples..

About Extensible Test Inputs

Test inputs are anything that the test designer identifies as needing validation. TestManager has two built-in test input types: requirements in the RequisitePro datastore, and elements in a Rose model. For more information about built-in test input types, see the *Using Rational TestManager* manual.

TestManager also supports extensible test inputs, which means that the test designer is not limited to using test inputs generated by Rational tools. Any kind of intermediary object needed for testing can be defined and managed as a test input type — for example, Microsoft Project files or Excel spreadsheets. Another example of a test input type might be C++ project files. These files might contain project-specific language restrictions and metrics, such as permitted depth of inheritance. The test designer could create test cases that determine whether specific source modules adhere to these restrictions.

For TestManager to support a new test input type, there must be a user-implemented DLL called a Test Input Adapter (TIA). The adapter includes functions for tasks such as connecting to and disconnecting from the test input source, testing whether an input element has been modified, and setting various kinds of filters.

Implementing a Test Input Adapter

TestManager requires a Test Input Adapter (TIA) for each test input type. There are TIAs already in place for RequisitePro requirements and Rational Rose models. For an extensible test input type, the customer must implement an API to support it. Certain functions must be included to make the adapter functional for tracking and reporting information about the connections between the test input and test cases. The following sections summarize these functions.

Connecting and Disconnecting

Your adapter must implement `TIConnect()` to connect to the test input source and `TIDisconnect()` to disconnect from the test input source. Your function to connect to the test input source might look something like the following example:

```
HRESULT TIConnect(const char ConnectInfo[TI_MAX_PATH], const char
UserID[TI_MAX_ID], char SourceID[TI_MAX_ID], char
ErrorDescription[TI_MAX_ERROR])
{
   HRESULT return_code = TI_SUCCESS;

   // Check to see whether connection information is valid.
   CFileStatus FileStatus;
   if (CFile::GetStatus(ConnectInfo, FileStatus) == TRUE)
   {
      try
      {
         // Open and connect to the datastore, using User ID if needed.

         // Store the connection context (information about the
         // connection) in a map or any other data structure.

         // Use ConnectInfo as the SourceID.
         _tcscpy(SourceID, ConnectInfo);
      }
      catch ()
      {
           return_code = TI_ERROR_UNABLE_TO_CONNECT;
      }
   }
   else
   {
      return_code = TI_ERROR_INVALID_CONNECTINFO;
   }
   return return_code;
}
```

TestManager can call `TIGetIsValidSource()` anytime after calling `TIConnect()` to determine whether the test source is still alive and accessible.

Your function to disconnect from the test input source might look like the following example:

```
HRESULT TIDisconnect(const char SourceID[TI_MAX_ID],
char ErrorDescription[TI_MAX_ERROR])
{
   HRESULT return_code = TI_SUCCESS;

   // Look up the context information for the specified SourceID.
   if (context)

   {
```

```
      try
      {
         // Delete the connection context from the map of connections.
      }
      catch (_com_error)
      {
         return_code = TI_ERROR_UNABLE_TO_DISCONNECT;
      }
   }
   else
   {
      return_code = TI_ERROR_INVALID_SOURCEID;
   }
   return return_code;
}
```

Organizing Test Input Elements

TestManager displays the structure of all test input sources in the Test Inputs window (from the **View** menu). Some test inputs are not hierarchically organized. For these files the `TIGetRoots()` function returns all of the test inputs in the source.

For test input types that are hierarchically organized, each input element of the test input source is represented as a node in the Test Input Tree. The TIA must, therefore, include a number of functions that enable TestManager to determine how the test input elements are organized. For these kinds of input sources, `TIGetRoots()` returns all of the root nodes. Once the root nodes have been established, other functions are called to build the Test Input Tree.

The `TIGetChildren()` function, for example, fills an array with the children of a specified parent node. The `TIGetChildren()` function might look like the following example:

```
HRESULT TIGetChildren(const char SourceID[TI_MAX_ID}, const char
NodeID[TI_MAX_ID]
struct Node *pChildNodes[], long *pNodeCount,
char ErrorDescription[TI_MAX_ERROR])
{
   HRESULT return_code = TI_SUCCESS;

   // Look up context information for the specified SourceID.

   if (context)
   {
      // Verify whether Node is valid.
      if (Node)
      {
         // Get children of node and store in an array, in this example
         // called PtrArray (used later).
      }
      else
```

```
      {
         return_code = TI_NODE_NOT_FOUND;
      }

      struct Node *pNodeArray;
      pNodeArray = new struct Node[PtrArray.GetSize()];


      for (long Index=0; Index < PtrArray.GetSize(); Index++)
      {
         // For each node set Name, ID, Type, IsOnlyContainer (Boolean)
         // and NeedsValidation (Boolean).
      }
      *pChildNodes = pNodeArray;
      *pNodeCount = PtrArray.GetSize();
   }
   else
   {
      return_count = TI_ERROR_INVALID_SOURCE_ID;
   }
   return return_code;
}
```

Other functions in the API for building the Test Input Tree are as follows:

- `TIGetIsChild()` – Determines whether an input element is a child node.
- `TIGetIsNode()` – Determines whether a specified test input element exists.
- `TIGetIsParent()` – Determines whether a test input element is a parent node.
- `TIGetParent()` – Finds the parent node of a specified test input element.
- `TIGetRoots()` – Fills an array with root elements extracted from the input source.
- `TIGetName()` – Extracts the name of a test input element.

Optional Functions

The test inputs API also includes the following functions that are useful, but not crucial, to the testing process:

- `TIGetIsModified()` – Determines whether an input element has been modified since the last call to `TIGetModified()`.
- `TIGetIsModifiedSince()` – Determines whether an input element has been modified since a specified date.
- `TIGetModified()` – Fills an array of structures with input elements that have been modified since the last call to `TIGetModified()`.

- `TIGetModifiedSince()` – Fills an array of structures with input elements that have been modified since a specified date.
- `TIGetNeedsValidation()`– Determines whether an input element requires validation.
- `TIGetSourceIcon()` – Points to the location of the bitmap file that contains the icon representing the test input source.
- `TIGetType()` – Extracts the name of the type of a test input element. This function is required if the test inputs are differentiated into more than one type.
- `TIGetTypes()` – Fills an array with an identifier for each type of test input element. This function is required if the test inputs are differentiated into more than one type.
- `TIGetTypeIcon()` – Points to the location of the icon that identifies nodes of a specified type.
- `TISetFilter()` – Filters operations towards input elements that meet specified criteria.
- `TISetValidationFilter()` – Filters operations according to validation status.
- `TIShowProperties()` – Displays the properties page of an input element.
- `TIShowSelectDialog()` – Displays a dialog box (provided by the adapter) for selecting elements from the input source.

You do not have to implement these functions, but doing so is highly recommended to enable access to all features of the current version and future versions of TestManager.

Building and Registering a New Adapter

After you have implemented the Test Input Adapter and created the DLL file, you compile the DLL using the **`__cdecl*`** calling convention. You then register the DLL with TestManager using the following procedure:

1. From TestManager click **Tools > Manage > Test Input Types.**
2. Click **New** in the Manage Test Inputs dialog box.
3. Enter the path to the DLL in the space provided for the adapter.

For detailed information about managing extensible test inputs, see the TestManager online Help.

Test Input Adapter Reference

6

This chapter describes the test inputs applications programming interface (API) in detail and also provides illustrations using code from the Rational RequisitePro adapter. For information about specific declarations, see the header file:

...Rational Test\rtsdk\c\include\testinputapi.h.

Summary of TIA functions

The Test Inputs Adapter (TIA) API includes the following functions:

Function	Description
`TIConnect()`	Connects to the test input source.
`TIDisconnect()`	Disconnects from the test input source.
`TIGetChildren()`	Fills an array with the children of a specified parent node.
`TIGetIsChild()`	Determines whether an input element is a child node.
`TIGetIsModified()`	Determines whether an input element has been modified since the last call to `TIGetModified()`.
`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()`.
`TIGetModifiedSince()`	Fills an array with input elements modified since a specified date.
`TIGetName()`	Extracts the user-readable name of an input element.

Function	Description
TIGetNeedsValidation()	Determines whether an input element requires validation.
TIGetParent()	Finds the parent node of an input element.
TIGetNode()	Retrieves the node for the specified input element.
TIGetRoots()	Fills an array with root elements extracted from the input source.
TIGetSourceIcon()	Points to the location of the bitmap file that contains the icon that is used for the source label.
TIGetType()	Extracts the name of the type of an input element.
TIGetTypeIcon()	Points to the location of the bitmap file of the icon that identifies nodes of a specified type.
TIGetTypes()	Fills an array with an identifier for each type of input element.
TISetFilter()	Filters display of test input elements.
TISetValidationFilter()	Filters operations according to validation status.
TIShowProperties()	Displays the property page of an input element.
TIShowSelectDialog()	Displays the selection dialog box 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 Inputs Tree.

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).

`IsOnlyContainer` set to `True` means that the input element cannot have test cases associated with it.

`NeedsValidation` set to `True` means that the input element needs to be tested.

Note on Memory Allocation

The TIA must allocate memory for all pointer to pointer parameters. TestManager deallocates it.

TIConnect()

Connects to the test input source.

Syntax

```
HRESULT TIConnect(const char ConnectInfo[TI_MAX_PATH], const
   char UserID[TI_MAX_ID], char SourceID[TI_MAX_ID], char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
ConnectInfo	INPUT. A string that specifies where the test inputs are located.
UserID	INPUT. A string that identifies the current tester.
SourceID	OUTPUT. A string that identifies the input source. The ID is used in subsequent calls to the adapter.
ErrorDescription	OUTPUT. A string for customized error messages.

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 input source specified in *ConnectInfo* is usually a path to the datastore. After the connection has been established, you assign a unique identifier for the input source to *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.

You can use the string *ErrorDescription* to provide the tester with additional information.

Example

```
//*********************************************************************
// TIConnect
//
//    Remarks:
//
//       This function creates a connection to the ReqPro Project
//       specified in the ConnectInfo parameter and returns a handle to
//       the client for subsequent calls to the adapter.
//
//    Parameters:
//
//       const char ConnectInfo[TI_MAX_PATH] [input]
//          Contains the path to the ReqPro Project RQS file.
//
//       const char UserID[TI_MAX_ID] [input]
//          The User ID of the user currently logged on.
//
//       char SourceID[TI_MAX_ID] [output]
//          A handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project for use by subsequent calls
//          to the adapter.
//
//       char ErrorDescription[TI_MAX_ERROR] [output]
//          An error description returned by the ReqPro COM server if
//          the connection to the ReqPro project fails.
//
//    Return Value:
//
//       TI_ERROR_INVALID_CONNECTINFO
//          The connection info was invalid - the RQS file didn't exist.
//
//       TI_ERROR_UNABLE_TO_CONNECT
//          The connection failed for some unknown reason.
//
//    Notes:
//
//       For the ReqPro adapter, the path to the RQS file is used as
```

```
//      the key to the connection map. The connection map associates
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the ReqPro Project.
//
//*********************************************************************
HRESULT TIConnect(char ConnectInfo[TI_MAX_PATH], const char
UserID[TI_MAX_ID], 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
         {

         /* 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.
         _tcscpy(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

`TIDisconnect()`

TIDisconnect()

Disconnects from the test input source.

Syntax

```
HRESULT TIDisconnect(const char SourceID[TI_MAX_ID], char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that TestManager uses to identify the connection.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCE_ID`. 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 operations are allowed on this input source.

You can use the string *ErrorDescription* to provide the tester with additional information.

Example

```
//**********************************************************************
// TIDisconnect
//
//    Remarks:
//
//       This function disconnects an active ReqPro Project connection.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          A handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
//       char ErrorDescription[TI_MAX_ERROR] [output]
//          An error description returned by the ReqPro COM server if
//          the method is unable to disconnect from the ReqPro Project.
//
//    Return Value:
//
//       TI_ERROR_INVALID_SOURCEID
//          There was no active connection for the specified source.
//
//       TI_ERROR_UNABLE_TO_DISCONNECT
//          The disconnection failed.
//
//    Notes:
//
//       For the ReqPro adapter, the path to the RQS file is used as
//       the key to the connection map. The connection map associates
//       an instance of CConnectionContext with each active connection.
//       Each instance of CConnectionContext stores critical
//       information about the connection to the ReqPro Project.
//
//**********************************************************************
HRESULT TIDisconnect(const 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. 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

`TIConnect()`

TIGetChildren()

Fills an array with the children of a specified parent node.

Syntax

```
HRESULT TIGetChildren(const char SourceID[TI_MAX_ID], const
   char NodeID[TI_MAX_ID], struct Node *PChildNodes[], long*
   pNodeCount, char ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the connection.
NodeID	INPUT. A string that identifies a parent node.
pChildNodes	OUTPUT. A pointer to a structure that receives the child elements of the node specified in *NodeID*.
pNodeCount	OUTPUT. A pointer to a long integer that is assigned the total number of child elements assigned to *pChildNodes*.
ErrorDescription	OUTPUT. A string for customized error messages.

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_SOURCE_ID` 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, you return an empty array.

You assign the total number of elements written to *`pChildNodes`* to *`pNodeCount`*.

You can use the string *`ErrorDescription`* to provide the user with additional information.

Example

```
//**********************************************************************
// TIGetChildren
//
//    Remarks:
//
//       This function returns all the child requirements of the
//       specified requirement.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          The handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
//       const char NodeID[TI_MAX_ID] [input]
//          The unique ID of the requirement; this method
//          returns its children.
//
//       struct Node *pChildNodes[]
//          An array of populated Node structures; each one defines
//          a child of the Node identified by the NodeID.
//          input parameter.
//
//       long* pNodeCount
//          The number of children returned by this method.
//
//       char ErrorDescription[TI_MAX_ERROR] [output]
//          An error description returned by the ReqPro COM server if
//          this method fails.
//
//    Return Value:
//
//       TI_ERROR_INVALID_SOURCEID
```

```
//         There was no active connection for the specified source.
//
//    Notes:
//
//       For the ReqPro adapter, the path to the RQS file is used as
//       the key to the connection map. The connection map associates
//       an instance of CConnectionContext with each active connection.
//       Each instance of CConnectionContext stores critical
//       information about the connection to the ReqPro Project.
//*********************************************************************
HRESULT TIGetChildren(const char SourceID[TI_MAX_ID], const char
NodeID[TI_MAX_ID], struct Node *pChildNodes[], long* pNodeCount, char
ErrorDescription[TI_MAX_ERROR])
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());

   HRESULT rc = TI_SUCCESS;

   struct Node *pNodeArray=0;
   *pNodeCount = 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 *)
         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.
         *pNodeCount = PtrArray.GetSize();
      }
      else
         rc = TI_ERROR_INVALID_SOURCEID;

      return rc;
   }
```

See Also

`TIGetParent()`

TIGetIsChild()

Determines whether an input element is a child node.

Syntax

```
HRESULT TIGetIsChild(const char SourceID[TI_MAX_ID], const char
   NodeID[TI_MAX_ID], BOOL* pIsChild, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
NodeID	INPUT. A string that identifies an input element.
pIsChild	OUTPUT. A pointer to a Boolean value that specifies whether the node is a child.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCE_ID`. The input source was incorrectly identified.
- `TI_ERROR_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

Set the Boolean value *pIsChild* to `True` if the element identified in *NodeID* is a child, and set the value to `False` if the element is not a child.

You can use the string *ErrorDescription* to provide the user with additional information.

See Also

`TIGetIsParent()`

TIGetIsModified()

Determines whether an input element has been modified since the last call to `TIGetModified()`.

Syntax

```
HRESULT TIGetIsModified(const char SourceID[TI_MAX_ID], const
   char NodeID[TI_MAX_ID], BOOL* pIsModified, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
NodeID	INPUT. A string that identifies an input element.
pIsModified	OUTPUT. A pointer to a Boolean value that specifies whether the input element has been modified.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCE_ID`. The input source was identified incorrectly.
- `TI_ERROR_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 *pIsModified* 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.

You can use the string *ErrorDescription* to provide the tester with additional information.

See Also

`TIGetModified()`, `TIGetIsModifiedSince()`

TIGetIsModifiedSince()

Determines whether an input element has been modified since a specified date.

Syntax

```
HRESULT TIGetIsModifiedSince(const char SourceID[TI_MAX_ID],
   const char NodeID[TI_MAX_ID], struct tm tmDate, BOOL*
   pIsModified, char ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
NodeID	INPUT. A string that identifies an input element.
tmDate	INPUT. The date, defined in time.h, which is part of the C-language Standard Library.
pIsModified	OUTPUT. A pointer to a Boolean value that specifies whether the input element has been modified.
ErrorDescription	OUTPUT. A string for customized error messages.

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_SOURCE_ID. The input source was identified incorrectly.
- TI_ERROR_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.

You can use the string *ErrorDescription* to provide the tester with additional information.

See Also

`TIGetIsModified()`

TIGetIsNode()

Determines whether a specified input element exists.

Syntax

```
HRESULT TIGetIsNode(const char SourceID[TI_MAX_ID], const char
   NodeID[TI_MAX_ID], BOOL* pIsNode, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
NodeID	INPUT. A string that identifies an input element.
pIsNode	OUTPUT. A pointer to a Boolean value that specifies whether the input element exists.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCE_ID`. The input source was identified incorrectly.
- `TI_ERROR_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

Set the Boolean value *pIsNode* to `True` if the element is there, and set the value to `False` if the element is not there.

You can use the string *ErrorDescription* to provide the tester with additional information.

Example

```
//*********************************************************************
// TIGetIsNode
//
//    Remarks:
//
//       This function indicates whether the specified value in NodeID
//       is a valid ReqPro Requirement in the source identified by
//       SourceID.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          A handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
//       const char NodeID [TI_MAX_ID] [input]
//          The ID of the requirement the caller (TestManager) wants to
//          verify exists.
//
//       BOOL* pIsNode [output]
//          Indicates whether NodeID represents a node that exists.
//
//       char ErrorDescription[TI_MAX_ERROR] [output]
//
//    Return Value:
//
//       TI_ERROR_INVALID_SOURCEID
//          There was no active connection for the specified source.
//
//    Notes:
//
//       For the ReqPro adapter, the path to the RQS file is used as
//       the key to the connection map. The connection map associates
//       an instance of CConnectionContext with each active connection.
//       Each instance of CConnectionContext stores critical
//       information about the connection to the ReqPro Project.
//*********************************************************************
HRESULT TIGetIsNode(const char SourceID[TI_MAX_ID], const char
NodeID[TI_MAX_ID], BOOL* pIsNode, char ErrorDescription[TI_MAX_ERROR])
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());

   HRESULT rc = TI_SUCCESS;
   *pIsNode = 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 *pIsNode 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 char SourceID[TI_MAX_ID], const
   char NodeID[TI_MAX_ID], BOOL* pIsParent, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
NodeID	INPUT. A string that identifies an input element.
pIsParent	OUTPUT. A pointer to a Boolean value that specifies whether the input element is a parent node in a hierarchical tree.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCE_ID`. The input source was identified incorrectly.
- `TI_ERROR_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

Set the Boolean value *pIsParent* to True if the specified input element (*NodeID*) is a parent, and set the value to False if the input element is not a parent.

You can use the string *ErrorDescription* to provide the tester with additional information.

Example

```
//*********************************************************************
// TIGetIsParent
//
//    Remarks:
//       This function indicates whether the specified NodeID
//       represents a requirement that has children.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          A handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
//       const char NodeID [TI_MAX_ID] [input]
//          The ID of the requirement for which the client wants to
//          determine the parental status.
//
//       BOOL* pIsParent [output]
//          Indicates whether NodeID represents a test input that has
//          children.
//
//       char ErrorDescription[TI_MAX_ERROR] [output]
//
//    Return Value:
//
//       TI_ERROR_INVALID_SOURCEID
//          There was no active connection for the specified source.
//
//    Notes:
//
//       For the ReqPro adapter, the path to the RQS file is used as
//       the key to the connection map. The connection map associates
//       an instance of CConnectionContext with each active connection.
//       Each instance of CConnectionContext stores critical
//       information about the connection to the ReqPro Project.
//*********************************************************************
HRESULT TIGetIsParent(const char SourceID[TI_MAX_ID], const char
NodeID[TI_MAX_ID], BOOL* pIsParent, 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 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()

Determines whether the specified test input source exists.

Syntax

`BOOL` **`TIGetIsValidSource`**`(const char` *`SourceID`*`[TI_MAX_ID])`

Element	Description
`SourceID`	INPUT. A string that identifies the 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.

See Also

`TIConnect()`, `TIDisconnect()`

TIGetModified()

Fills an array of structures with input elements that have been modified since the last call to `TIGetModified()`.

Syntax

```
HRESULT TIGetModified(const char SourceID[TI_MAX_ID], struct
   Node ModifiedNodes[], long* pNodeCount, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
ModifiedNodes	OUTPUT. An array of structures that receives elements that have been modified.
pNodeCount	OUTPUT. A pointer to a long integer that specifies the number of elements assigned to *ModifiedNodes*.
ErrorDescription	OUTPUT. A string for customized error messages.

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_SOURCE_ID` if 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 *pNodeCount*.

You can use the string *ErrorDescription* to provide the tester with additional additional information.

See Also

`TIGetIsModified()`, `TIGetModifiedSince()`

TIGetModifiedSince()

Fills an array of node structures with input elements modified since a specified date.

Syntax

```
HRESULT TIGetModifiedSince(const char SourceID[TI_MAX_ID],
   struct tm tmDate, struct Node ModifiedNodes[], long*
   pNodeCount, char ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
tmDate	INPUT. The date, defined in time.h, which is part of the C-language Standard Library.
ModifiedNodes	OUTPUT. An array of structures that receives the elements modified since *tmDate*.
pNodeCount	OUTPUT. A pointer to a long integer that specifies the total number of elements written to *ModifiedNodes*.
ErrorDescription	OUTPUT. A string for customized error messages.

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_SOURCE_ID` if 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 *pNodeCount*.

You can use the string *ErrorDescription* to provide the tester with additional information.

See Also

`TIGetModified(), TIGetIsModified(), TIGetIsModifiedSince()`

TIGetName()

Extracts the name of an input element.

Syntax

```
HRESULT TIGetName(const char SourceID[TI_MAX_ID], const char
   NodeID[TI_MAX_ID], char Name[TI_MAX_NAME], char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
NodeID	INPUT. A string that identifies an input element.
Name	OUTPUT. A string that specifies the name of the input element.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCE_ID`. The input source was identified incorrectly.
- `TI_ERROR_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*.

You can use *ErrorDescription* to provide the tester with additional information.

Example

```
//**********************************************************************
// TIGetName
//
//    Remarks:
//       This function returns the name of the requirement
//       identified by the value of NodeID.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          The handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
//       const char NodeID[TI_MAX_ID] [input]
//          The unique ID of the requirement; this method
//          returns its name.
//
//       char Name[TI_MAX_NAME] [output]
//          The name of the test input.
//
//       char ErrorDescription[TI_MAX_ERROR] [output]
//          An error description returned by the ReqPro COM server if
//          this method is unable to obtain the name.
//
//    Return Value:
//
//       TI_NODE_NOT_FOUND
//          The specified NodeID does not exist.
//
//       TI_ERROR_INVALID_SOURCEID
//          There was no active connection to the specified source.
//
//    Notes:
//
//       For the ReqPro adapter, the path to the RQS file is used as
//       the key to the connection map. The connection map associates
//       an instance of CConnectionContext with each active connection.
//       Each instance of CConnectionContext stores critical
//       information about the connection to the ReqPro Project.
//**********************************************************************
HRESULT TIGetName(const char SourceID[TI_MAX_ID], const char
NodeID[TI_MAX_ID], char Name[TI_MAX_NAME], 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 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

TIGetIsNode(), TIGetNeedsValidation()

TIGetNeedsValidation()

Determines whether an input element requires validation.

Syntax

```
HRESULT TIGetNeedsValidation(const char SourceID[TI_MAX_ID],
   const char NodeID[TI_MAX_ID], BOOL* pNeedsValidation, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
NodeID	INPUT. A string that identifies an input element.
pNeedsValidation	OUTPUT. A pointer to a Boolean value that specifies whether the input element requires validation.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns one of the following values:

- `TI_SUCCESS`. The function completed successfully.
- `TI_ERROR_INVALID_SOURCE_ID`. The input source was identified incorrectly.
- `TI_ERROR_NODE_NOT_FOUND`. The adapter was unable to locate the specified input element.

Comments

Set the value of Boolean *`pNeedsValidation`* 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.

You can use the string *`ErrorDescription`* to provide the tester with additional information.

See Also

`TIGetIsNode(), TIGetName(), TISetValidationFilter()`

TIGetNode()

Retrieves the node of the specified input element.

Syntax

```
HRESULT TIGetNode(const char SourceID[TI_MAX_ID], const char
   NodeID[TI_MAX_ID], struct Node **pNode, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
`SourceID`	INPUT. A string that identifies the connection.
`NodeID`	INPUT. A string that identifies a parent node.
`pNode`	OUTPUT. A pointer to the node specified in *`NodeID`*.
`ErrorDescription`	OUTPUT. A string for customized error messages.

Return Values

This function typically returns `TI_SUCCESS` when the function completes successfully and returns `TI_ERROR_INVALID_SOURCE_ID` 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;
```

You can use the string *`ErrorDescription`* to provide the user with additional information.

See Also

`TIGetIsNode()`

TIGetParent()

Finds the parent node of an input element.

Syntax

```
HRESULT TIGetParent(const char SourceID[TI_MAX_ID], const char
   NodeID[TI_MAX_ID], struct Node** pParentNode, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
`SourceID`	INPUT. A string that identifies the input source.
`NodeID`	INPUT. A string that identifies an input element.
`pParentNode`	OUTPUT. A pointer to a structure that receives the parent node.
`ErrorDescription`	OUTPUT. A string for customized error messages.

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_SOURCE_ID`. The input source was identified incorrectly.
- `TI_ERROR_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.

You can use the string *`ErrorDescription`* to provide the tester with additional information.

Example

```
//**********************************************************************
// TIGetParent
//
//    Remarks:
//       This function returns the parent of the specified requirement.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          The handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
//       const char NodeID[TI_MAX_ID] [input]
//          The unique ID of the requirement; this method
//          returns a populated Node structure with information about
//          its parent.
//
//       struct Node **pNode [output]
//          A pointer to a populated Node structure containing the
//          information about the parent requirement.
//
//       char ErrorDescription[TI_MAX_ERROR] [output]
//          An error description returned by the ReqPro COM server if
//          this method is unable to obtain the parent of the specified
//          requirement.
//
//    Return Value:
//
//       TI_NODE_NOT_FOUND
//          The specified NodeID does not exist.
//
```

```
//      TI_ERROR_INVALID_SOURCEID
//         There was no active connection to the specified source.
//   Notes:
//
//      For the ReqPro adapter, the path to the RQS file is used as
//      the key to the connection map. The connection map associates
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the ReqPro Project.
//*********************************************************************
HRESULT TIGetParent(const char SourceID[TI_MAX_ID], const char
NodeID[TI_MAX_ID], struct Node **pParentNode, 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 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);

      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 char SourceID[TI_MAX_ID], struct Node
   *pRootNodes[], long *pNodeCount, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
pRootNodes	OUTPUT. A string of structures of type Node that receives the root nodes.
pNodeCount	OUTPUT. A pointer to a long integer that specifies the total number of root nodes.
ErrorDescription	OUTPUT. A string for customized error messages.

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_SOURCE_ID. The input source was identified incorrectly.
- TI_ERROR_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*.

You can use the string *ErrorDescription* to provide the tester with additional information.

Example

```
//**********************************************************************
// TIGetRoots
//
//    Remarks:
//
//       This function returns all the root nodes for this RequisitePro
//       Project. Because ReqPro supports hierarchical organization of
//       requirements, this method only returns a subset of all
//       requirements. However, for test input sources that do not
//       support hierarchical organization, this function returns all
//       the test inputs in the source — subject to any filtering that
//       might be offered by the adapter.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          The handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
//       struct Node * pRootNodes [] [output]
//
//          An array of populated Node structures; each one defines a
//          requirement.
//
//       long* pNodeCount [output]
//
//          The number of nodes (that is, test inputs) returned by this
//          method.
//
//       char ErrorDescription[TI_MAX_ERROR] [output]
//          An error description returned by the ReqPro COM server if
//          this method cannot obtain the root requirement.
//
//    Return Value:
//
//       TI_ERROR_INVALID_SOURCEID
//          There was no active connection to the specified source.
//
//    Notes:
//
//       For the ReqPro adapter, the path to the RQS file is used as
```

```
//      the key to the connection map. The connection map associates
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the ReqPro Project.
//*************************************************************************
HRESULT TIGetRoots(const char SourceID[TI_MAX_ID], struct Node
*pRootNodes[], long* pNodeCount, char 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.
      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.
         *pNodeCount = PtrArray.GetSize();
      }
      else
         rc = TI_ERROR_INVALID_SOURCEID;

      return rc;
   }
```

See Also

TIGetIsNode(), TIGetIsParent(), TIGetParent()

TIGetSourceIcon()

Points to the location of the graphics file (16 x 16 bitmap) that contains the icon that is used to identify the input source.

Syntax

HRESULT **TIGetSourceIcon**(const char *SourceID*[TI_MAX_ID], char *IconPath*[TI_MAX_PATH], char *ErrorDescription*[TI_MAX_ERROR])

Element	Description
SourceID	INPUT. A string that identifies the input source.
IconPath	OUTPUT. A string that identifies the location of the icon graphics file.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns `TI_SUCCESS` when the function completes successfully and returns `TI_ERROR_INVALID_SOURCE_ID` if 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.

You can use the string *`ErrorDescription`* to provide the tester with additional information.

Example

```
//**********************************************************************
// TIGetSourceIcon
//
//    Remarks:
//
//       This function returns the path to a 16 x 16 BMP file, which
//       contains the icon that represents a RequisitePro test input
//       source.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          The handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
//       char IconPath[TI_MAX_PATH] [output]
//
//          The path to the BMP, which contains the Icon for the
//          test input source Node structure.
//
//       char ErrorDescription[TI_MAX_ERROR] [output]
//
//       Notes:
//       For the ReqPro adapter, the source icon is the yellow pyramid
//       that the ReqPro executable uses as its icon.
//**********************************************************************
HRESULT TIGetSourceIcon(const char SourceID[TI_MAX_ID], char
IconPath[TI_MAX_PATH], char 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.
   _tcscpy(szModuleFilePath, szDrive);
   _tcscat(szModuleFilePath, szDir);
   _tcscat(szModuleFilePath, "bitmap_source.bmp");

   // Copy the path into the return variable.
   _tcscpy(IconPath, szModuleFilePath);

   return TI_SUCCESS;
}
```

See Also

TIGetTypeIcon()

TIGetType()

Extracts the name of the type of an input element.

Syntax

```
HRESULT TIGetType(const char SourceID[TI_MAX_ID], const char
   NodeID[TI_MAX_ID], char Type[TI_MAX_TYPE], char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
NodeID	INPUT. A string that identifies an input element.
Type	OUTPUT. A string that identifies the type of node.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns one of the following values:

- TI_SUCCESS. The function completed successfully.
- TI_ERROR_INVALID_SOURCE_ID. The input source was identified incorrectly.

- `TI_ERROR_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`*.

You can use the string *`ErrorDescription`* to provide the tester with additional information.

Example

```
//*********************************************************************
// TIGetType
//
//    Remarks:
//
//       This function returns the type of the requirement identified
//       by the value of NodeID. For some test input adapters,
//       there may be only one type of input element.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          The handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
//       const char NodeID[TI_MAX_ID] [input]
//          The unique ID of the requirement; this method returns its
//          type.
//
//       char Type[TI_MAX_NAME] [output]
//          The type of the RequisitePro requirement.
//
//       char ErrorDescription[TI_MAX_ERROR] [output]
//          An error description returned by the ReqPro COM server if
//          this method is unable to obtain the type of the requirement.
//
//    Return Value:
//
//       TI_SUCCESS
//          The function completed successfully.
//
//       TI_NODE_NOT_FOUND
//          The specified NodeID does not exist.
//
//       TI_ERROR_INVALID_SOURCEID
//          There was no active connection to the specified source.
//
//    Notes:
//       For the ReqPro adapter, the path to the RQS file is used as
//       the key to the connection map. The connection map associates
```

```
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the ReqPro Project.
//*****************************************************************
HRESULT TIGetType(const char SourceID[TI_MAX_ID], const char
NodeID[TI_MAX_ID], char Type[TI_MAX_NAME], 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 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 char SourceID[TI_MAX_ID], const
   char Type[TI_MAX_TYPE], char IconPath[TI_MAX_PATH], char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
Type	INPUT. A string that identifies the type of node.
IconPath	OUTPUT. A string that specifies the location of a graphics file for an icon that represents nodes of a particular type.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns `TI_SUCCESS` when the function completes successfully and returns `TI_ERROR_INVALID_SOURCE_ID` 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*.

You can use the string *ErrorDescription* to provide the tester with additional information.

Example

```
//*********************************************************************
// TIGetTypeIcon
//
//    Remarks:
//
//       This function returns the path to a 16 x 16 BMP file that
//       contains the icon representing a RequisitePro requirement of a
//       particular type. For ReqPro, all requirement types use the
//       same icon. This will not be true for other test input types,
//       such as Rational Rose.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          The handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
```

```
//      char Type [TI_MAX_TYPE] [input]
//         The type of the test input; the client is
//         requesting the associated icon.
//
//      char IconPath[TI_MAX_PATH] [output]
//         The path to the BMP that contains the icon for the specified
//         type.
//
//      char ErrorDescription[TI_MAX_ERROR] [output]
//*****************************************************************
HRESULT TIGetTypeIcon(const char SourceID[TI_MAX_ID], const char
Type[TI_MAX_TYPE], char IconPath[TI_MAX_PATH], char
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.
   _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()

Fills an array with an identifier for each type of input element.

Syntax

```
HRESULT TIGetTypes(const char SourceID[TI_MAX_ID], char
   (**Types)[TI_MAX_TYPE], long* pTypeCount, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
Types	OUTPUT. A pointer to an array of type identifiers.
pTypeCount	OUTPUT. A pointer to a long integer that specifies the total number of test types.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns TI_SUCCESS when the function completes successfully and returns TI_ERROR_INVALID_SOURCE_ID 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 *pTypeCount*.

You can use the string *ErrorDescription* to provide the tester with additional information.

Example

```
//*********************************************************************
// TIGetTypes
//
//    Remarks:
//       This function returns all the requirement types in the
//       RequisitePro project.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          The handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
//       char (** Types)[TI_MAX_TYPE] [output]
//          An array of character strings; each represents a
```

```
//         unique requirement type (that is, test input type).
//
//      long* plTypeCount [output]
//
//         The number of requirement types returned by this method.
//
//      char ErrorDescription[TI_MAX_ERROR] [output]
//
//         An error description returned by the ReqPro COM server if
//         this method fails.
//
//   Return Value:
//
//      TI_ERROR_INVALID_SOURCEID
//         There was no active connection to the specified source.
//
//   Notes:
//
//      For the ReqPro adapter, the path to the RQS file is used as
//      the key to the connection map. The connection map associates
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the ReqPro Project.
//*************************************************************************
HRESULT TIGetTypes(const char SourceID[TI_MAX_ID], char (**
Types)[TI_MAX_TYPE], long* plTypeCount, char
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.
               _tcscpy((*pszTypes)[lIndex-1], (char
               *)MyReqTypePtr->GetName());
            }
            // Store count of types in return variable.
            *plTypeCount = MyReqTypesPtr->GetCount();
         }
         catch (_com_error)
         {
            rc = TI_ERROR;
         }
      }
      else
         rc = TI_ERROR_INVALID_SOURCEID;

      return rc;
   }
```

See Also

TIGetType(), TIGetTypeIcon()

TISetFilter()

Filters the display of test input elements.

Syntax

HRESULT **TISetFilter**(const char *SourceID*[TI_MAX_ID], const char *UserID*[TI_MAX_ID], char *ErrorDescription*[TI_MAX_ERROR])

Element	Description
SourceID	INPUT. A string that identifies the input source.

Element	Description
UserID	INPUT. A string that identifies the current user.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns TI_SUCCESS when the function completes successfully and returns TI_ERROR_INVALID_SOURCE_ID if the input source was identified incorrectly.

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.

You can use the string *ErrorDescription* to provide the tester with additional information.

Example

```
//**********************************************************************
// TISetFilter
//
//    Remarks:
//
//       This function displays a dialog that enables the tester to
//       filter on specific requirement types. The filter settings are
//       stored in the Registry. These settings are specific to each
//       user. This method is optional.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          The handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
//       const char UserID[TI_MAX_ID] [input]
//          UserName of currently logged in user.
//
//       char ErrorDescription[TI_MAX_ERROR] [output]
//
//    Return Value:
//
//       TI_ERROR_INVALID_SOURCEID
//          There was no active connection to the specified source.
//
```

```
//    Notes:
//       For the ReqPro adapter, the path to the RQS file is used as
//       the key to the connection map. The connection map associates
//       an instance of CConnectionContext with each active connection.
//       Each instance of CConnectionContext stores critical
//       information about the connection to the ReqPro Project.
//*********************************************************************
HRESULT TISetFilter(const char SourceID[TI_MAX_ID], const char
UserID[TI_MAX_ID], char 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)
      {

      }
   }
   else
      rc = TI_ERROR_INVALID_SOURCEID;

   return rc;
}
```

See Also

TISetTypeFilter(), TISetValidationFilter()

TISetValidationFilter()

Filters operations according to validation status.

Syntax

```
HRESULT TISetValidationFilter(const char SourceID[TI_MAX_ID],
   const char UserID[TI_MAX_ID], BOOL OnlyNeedsValidation, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
UserID	INPUT. A string that identifies the current user.
OnlyNeedsValidation	INPUT. A pointer to a Boolean value that specifies whether filtering by validation status is in effect.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns `TI_SUCCESS` when the function completes successfully and returns `TI_ERROR_INVALID_SOURCE_ID` if the input source was identified incorrectly.

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 *OnlyNeedsValidation* is set to `False`, filtering by this criterion is disabled.

You can use the string *ErrorDescription* to provide the tester with additional information.

See Also

`TISetFilter()`, `TISetTypeFilter()`

TIShowProperties()

Displays the property page of an input element.

Syntax

```
HRESULT TIShowProperties(char const SourceID[TI_MAX_ID], const
   char NodeID[TI_MAX_ID], struct Node** pModifiedNode, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.
NodeID	INPUT. A string that identifies an input element.
pModifiedNode	OUTPUT. A pointer to a structure that receives the new information about a node.
ErrorDescription	OUTPUT. A string for customized error messages.

Return Values

This function typically returns one of the following values:

- TI_SUCCESS. The function completed successfully.
- TI_ERROR_INVALID_SOURCE_ID. The input source was identified incorrectly.
- TI_ERROR_NODE_NOT_FOUND. The adapter was unable to locate the specified input element.

Comments

If any property relevant to TestManager has changed, assign the new information to a Node structure associated with the pointer *pModifiedNode*.

You can use the string *ErrorDescription* to provide the tester with additional information.

Example

```
//*********************************************************************
// TIShowProperties
//
//    Remarks:
//
//       This function displays the properties of the specified
//       requirement and returns specific information (for example,
//       name and type) about the requirement. This function is
//       optional.
//       The properties dialog is provided by the RequisitePro COM
```

```
//      server.
//
//   Parameters:
//
//      const char SourceID[TI_MAX_ID] [input]
//         The handle, provided by the adapter, that identifies the
//         connection to the ReqPro Project.
//
//      const char NodeID[TI_MAX_ID] [input]
//         The unique ID of the requirement; this method
//         returns a populated Node structure.
//
//      struct Node ** pModifiedNode [output]
//         A pointer to a populated Node structure containing the
//         information about the test input identified by NodeID. It
//         must contain any change that might have been made by using
//         the Property dialog box.
//
//      char ErrorDescription[TI_MAX_ERROR] [output]
//         An error description returned by the ReqPro COM server if
//         this method is unable to display the properties dialog.
//
//   Return Value:
//      TI_NODE_NOT_FOUND
//         The specified NodeID does not exist.
//
//      TI_ERROR_INVALID_SOURCEID
//         There was no active connection for the specified source.
//
//   Notes:
//      For the ReqPro adapter, the path to the RQS file is used as
//      the key to the connection map. The connection map associates
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the ReqPro Project.
//*********************************************************************
HRESULT TIShowProperties(const char SourceID[TI_MAX_ID], const char
NodeID[TI_MAX_ID], struct Node** pModifiedNode, 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 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()

Displays a selection dialog box for choosing elements from the input source.

Syntax

```
HRESULT TIShowSelectDialog(const char SourceID[TI_MAX_ID],
   struct Node* pSelectedNodes[], long* pNodeCount, char
   ErrorDescription[TI_MAX_ERROR])
```

Element	Description
SourceID	INPUT. A string that identifies the input source.

Element	Description
pSelectedNodes	OUTPUT. An array of structures that specifies the selected input elements.
pNodeCount	OUTPUT. A pointer to a long integer that specifies the total number of nodes selected.
ErrorDescription	OUTPUT. A string for customized error messages (optional).

Return Values

This function typically returns `TI_SUCCESS` when the function completes successfully and returns `TI_ERROR_INVALID_SOURCE_ID` if the input source was identified incorrectly.

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 *pNodeCount*.

You can use the string *ErrorDescription* to provide the tester with additional information.

Example

```
//*********************************************************************
// TIShowSelectDialog
//
//    Remarks:
//
//       This function displays a dialog that enables a tester to select
//       one or more requirements. It is used by TestManager when
//       associating test inputs to test cases. This function is
//       optional.
//
//    Parameters:
//
//       const char SourceID[TI_MAX_ID] [input]
//          The handle, provided by the adapter, that identifies the
//          connection to the ReqPro Project.
//
//       struct Node * pSelectedNodes [output]
//          An array of populated Node structures; each one defines a
//          test input.
//
```

```
//      long* pNodeCount [output]
//         The number of nodes (that is, requirements) returned by this
//         method.
//
//      char ErrorDescription[TI_MAX_ERROR] [output]
//         An error description returned by the ReqPro COM server if
//         this method is unable to display the dialog.
//
//   Return Value:
//
//      TI_NODE_NOT_FOUND
//         The specified NodeID does not exist.
//
//      TI_ERROR_INVALID_SOURCEID
//         There was no active connection to the specified source.
//
//   Notes:
//
//      For the ReqPro adapter, the path to the RQS file is used as
//      the key to the connection map. The connection map associates
//      an instance of CConnectionContext with each active connection.
//      Each instance of CConnectionContext stores critical
//      information about the connection to the ReqPro Project.
//*************************************************************************
HRESULT TIShowSelectDialog(const char SourceID[TI_MAX_ID], struct Node
*pSelectedNodes[], long* pNodeCount, char
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()

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V