Rational Software Corporation

Customizing Rational Rose RealTime for Target Control and Observability

RATIONAL ROSE® REALTIME

VERSION: 2002.05.20

WINDOWS/UNIX

Rational®
the software development company

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

Part Number: 800-025105-000

Version Number: 2002.05.20

Contents

Customizing for Target Control and Observability 1

Contents

This chapter is organized as follows:

Introduction

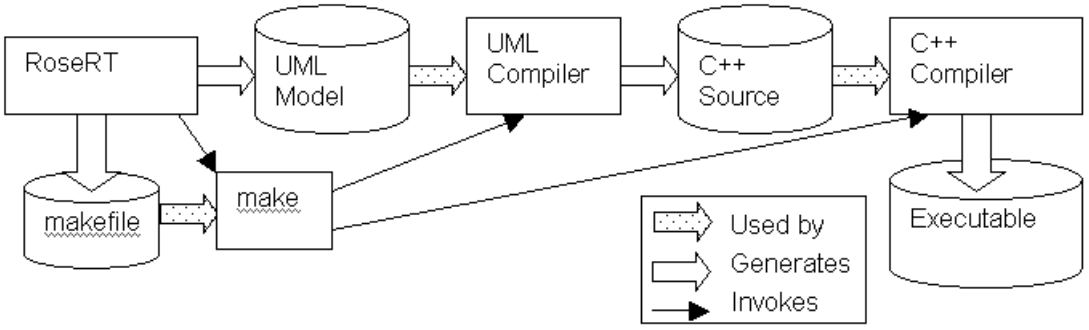
Rational Rose RealTime is a comprehensive visual modeling environment that delivers a powerful combination of notation, processes, and tools optimized to meet the challenges of real-time software development. The Rational Rose RealTime UML model compiler converts models directly into executable applications. Those executables can be controlled and debugged at run-time under the control of the toolset. Rational Rose RealTime integrates with source debuggers providing the developer with the choice of debugging at the UML and source code level. A combination of UML editors, a model compiler, and run-time debugging tools address the complete life-cycle of a project from early use case analysis through design, implementation, and test.

This document describes how to add support to Rational Rose RealTime 6.0 and later for target control and observability, and how to integrate Rational Rose RealTime with source code debuggers.

Model Compilation and Target Control

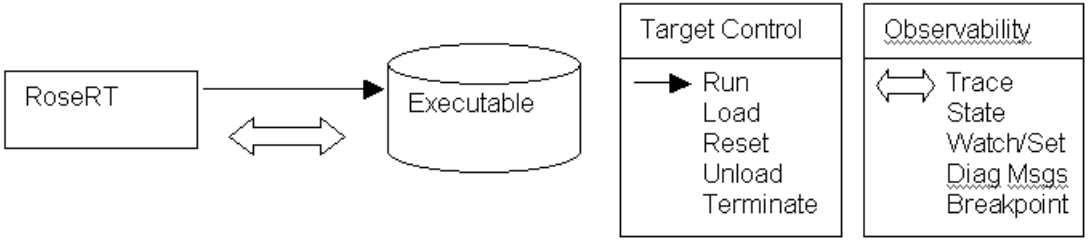
Rational Rose RealTime models are compiled seamlessly into applications ready for execution on the host or target operating systems. Figure 1 provides a high level overview of model compilation.

Figure 1 UML Model Compilation



Rational Rose RealTime also has the ability to control the executing application at run-time (for example, during debugging). Target Observability provides the ability to observe and debug the executing application at the UML level. Figure 2 shows a simplified high-level overview of Target Control and Observability.

Figure 2 Target Control and Observability



Rational Rose RealTime also supports inter-working with traditional source code debuggers. This enables developers to control, observe, and debug the application at the UML level and detailed source code level simultaneously.

Intended Audience

This guide is specifically designed for technical staff responsible for enabling these capabilities for a specific target execution environment. It is assumed that the reader has significant knowledge and experience with the development environment, operating system, and target hardware.

Target Control

Target Control refers to the Rational Rose RealTime toolset's features that load, unload, execute, and terminate a Rational Rose RealTime-generated application, as well as the ability to reset a remote target platform.

Target Control is not the same feature as Target Observability. Target Observability allows the observation of the application executing on a target from the UML level (such as state change, state machine breakpoints, event tracing, and so on) on the host-based toolset. Target Control interacts with the APIs of the target execution environment to load, run, and terminate the application, whereas Target Observability communicates directly with the running application.

Target Control Modes

Rational Rose RealTime supports three different Target Control modes:

- Manual Mode
- Basic Mode
- Debugger Mode

Manual Mode

In **Manual mode,** Rational Rose RealTime does not provide any Target Control functionality. The user is responsible for performing Target Control operations (such as loading, executing). After the target application starts, the user can direct the Rational Rose RealTime toolset to connect to the executing target application for Target Observability.

Basic Mode

In **Basic mode**, Rational Rose RealTime uses the target environment's APIs to control the execution of the target application. Rational Rose RealTime supports automatic target control for a number of host and target platform combinations. Users deploy on a number of other target environments as well.

Rational Rose RealTime uses Perl scripts to perform the Target Control operations. These scripts can call the target APIs directly or can call some intermediary helper application to control the execution on the target.

There are five Target Control scripts:

- reset.pl
- load.pl
- unload.pl
- execute.pl
- terminate.pl

Debugger Mode

Debugger mode provides same the capabilities as Basic mode and, in addition, provides the ability to inter-work with a C or C++ source debugger (for example, Visual C++) to set source code level breakpoints from within the UML model. When these source breakpoints are hit at run-time, control of the executable is passed to the source debugger. When the application is continued, control of the executable is passed back to the Rational Rose RealTime toolset. Debugger mode provides an integrated debug environment that permits a simultaneous use of source code and UML debugging styles.

Target Control Scripts

When you open the Specification dialog for a **Processor** in the **Deployment View**, notice that **Load Scripts** text box specifies the path to the Target Control scripts (for example, $TARGET_PATH/win32/, $TARGET_PATH/tornado2/). This directory contains up to five Target Control scripts, each of which has a different function:

- **reset.pl** - resets the target processor, see Reset
- **load.pl** - loads a Component onto a target, see Load
- **unload.pl** - unloads a Component from a target, see Unload
- **execute.pl** - executes a Component, see Execute
- **terminate.pl** - terminates the execution of a Component, see Terminate

The Target Control Scripts determine the Target Control capabilities for the Processor. If a script exists in the Target Control Scripts directory, then the toolset assumes that the corresponding capability exists. Whenever a Component Instance is created on a Processor (that is, a Component in the **Component View** is assigned to a Processor in the **Deployment View**), the toolset checks to see which scripts are available and enables those capabilities in the toolset menus that are accessible by right-clicking on a Component Instance. These menu options are now available to the user.

The presence of the scripts is not their only purpose. Each existing Target Control script must also provide the associated capability. For example, the load script must load the corresponding component onto the target specified by the Processor, and so on. The scripts use information from Processor and Component Instances specifications, but note that the scripts do not need to use all the parameters that are passed to them. Any script just needs to process the arguments that allow it to perform its intended operation.

These scripts are written in Perl, but they may spawn other executables that may be needed to provide the desired capability. Every script also indicates whether it was successful.

Menu Commands

If the path to the Target control scripts contains the following scripts, that corresponding menu command will become active on the **Processor** menu:

- **reset.pl** - resets the target processor and activates the **Reset** menu option
- **load.pl** - loads a Component onto a target and activates the **Load** menu option
- **unload.pl** - unloads a Component from a target and activates the **Unload** menu option
- **execute.pl** - executes a Component and activates the Run menu option (**Execute)**
- **terminate.pl** - terminates the execution of a Component and activates the **Shutdown** menu option (**Terminate**)

Reset

Description

The reset.pl script resets a target processor. If this script exists, the **Reset** menu item will be active on the corresponding Processor menu.

Command Line

Rtperl reset.pl –ip *target* –**server** *targetServer* –**os** *targetOS* –**cpu** *targetCPU*

Arguments

-ip ***target***	Target name or address.
-server ***targetServer***	Target server name or address.
-os ***OS***	OS executing on target.
-cpu ***CPU***	CPU on the target.

Returns

::Ok::	String indicating success.
Error String	Error string to be displayed in error message box in the toolset.

Note: The data for the script arguments are retrieved from the **Processor Specification** dialog.

Load

Description

The load.pl script loads a component onto the corresponding target processor. If this script exists, the **Load** menu item will be active on the corresponding Component Instance menu when the Component Instance is in a "loadable" state.

Command Line

Rtperl load.pl –ip *target* –**server** *targetServer* –**os** *targetOS* –**cpu** *targetCPU*

-exe *componentDir* –**prio** *priority* –**port** *Toport*

Arguments

-ip ***target***	Target name or address.
-server ***targetServer***	Target server name or address.
-os ***OS***	OS executing on target.
-cpu ***CPU***	CPU on the target.
-exe ***executable***	6.1 and later: Fully qualified executable name.
-prio ***priority***	Priority to run the component instance
-port ***Toport***	Target Observability port.

Returns

::Ok:: [-warning 'xxx'] [-passback xxx]	6.1 and later: String indicating success. Now two option parameters may follow the **::Ok:: string: -warning** and **-passback**. See General Issues.
Error String	Error string to be displayed in error message box in the toolset.

Note: The data for the options are retrieved from the **Processor** and **Component Instance Specification**.

Unload

Description

The unload.pl script removes a component from the corresponding target processor. If this script exists, the **Unload** menu item will be active on the corresponding Component Instance menu when the Component Instance is in an "unloadable" state.

Command Line

Rtperl unload.pl –ip *target* –**server** *targetServer* –**os** *targetOS* –**cpu** *targetCPU*

-exe *componentDir* –**prio** *priority* –**port** *TOport* **paramsFromLoad**

Arguments

-ip ***target***	Target name or address.
-server ***targetServer***	Target server name or address.
-os ***OS***	OS executing on target.
-cpu ***CPU***	CPU on the target.
-exe ***executable***	6.1 and later: Fully qualified executable name.
-prio ***priority***	Priority to run the component instance
-port ***Toport***	Target Observability port.
ParamsFromLoad	Any parameters that were returned from a successful **Load** operation.

Returns

::Ok:: [-warning 'xxx']	6.1 and later: String indicating success. Now, one option parameter may follow **::Ok:: string: -warning**. See General Issues.
Error String	Error string to be displayed in error message box in the toolset.

Note: The data for the options are retrieved from the **Processor** and **Component Instance Specification**.

Execute

Description

The execute.pl script starts execution of a component instance on the corresponding target processor. If this script exists, the **Run** menu item is available on the Component Instance menu when the Component Instance is in a "runable" state.

Command Line

Rtperl execute.pl –ip *target* –**server** *targetServer* –**os** *targetOS* –**cpu** *targetCPU*

-exe *componentDir* –**prio** *priority* –**port** *Toport*

-args *commandLineArgs*

Arguments

-ip ***target***	Target name or address.
-server ***targetServer***	Target server name or address.
-os ***OS***	OS executing on target.
-cpu ***CPU***	CPU on the target.
-exe ***componentDir***	6.0.x: Path to Component directory. It is used to locate the component.
-exe ***executable***	6.1 and later: Fully qualified executable name.
-prio ***priority***	Priority to run the component instance
-port ***Toport***	Target Observability port.
-args ***commandLineArgs***	Command Line arguments that are to be used when starting the target application. Parameters that follow the **-args** tag are all passed to the target application.

Returns

:Ok:: paramsFromExecute	String indicating success. Any strings passed back after the **::Ok::** will be based to the terminate.pl script when the user invokes the *Shutdown* command.
::Ok:: [-warning 'xxx'] [-passback xxx]	6.1 and later: String that represents the operation was successful. Now two option parameters may follow the **::Ok::** string: -**warning** and -**passback**. See General Issues.
Error String	Error string to be displayed in error message box in the toolset.

Note: The data for the options are retrieved from the Processor and Component Instance Specification.

An example of **paramsFromExecute** is a handle that identifies the process that was created. For example, on Windows we return **–pid nnnnnn.** This allows us to pass back the PID (Process ID) to the Terminate script.

Terminate

Description

The terminate.pl script is used to kill a component instance on the corresponding target processor. If this script exists, the **Shutdown** menu item will be active on the corresponding Component Instance menu when the Component Instance is in a "killable" state.

Command Line

Rtperl execute.pl –ip *target* –**server** *targetServer* –**os** *targetOS* –**cpu** *targetCPU*

-exe *componentDir* –**prio** *priority* –**port** *TOport*
paramsFromExecute

Arguments

-ip ***target***	Target name or address.
-server ***targetServer***	Target server name or address.
-os ***OS***	OS executing on target.
-cpu ***CPU***	CPU on the target.

-exe *executable*	6.1 and later: Fully qualified executable name.
-prio *priority*	Priority to run the component instance.
-port *Toport*	Target Observability port.
ParamsFromExecute	Any parameters that were returned from a successful **Run** operation.

Returns

::Ok:: [-warning 'xxx']	6.1 and later: String indicating success. Now optional parameter may follow the **::Ok:: string: -warning**. See General Issues.
Error String	Error string to be displayed in error message box in the toolset.

Note: The data for the options are retrieved from the **Processor** and **Component Instance Specification**.

General Issues

- The –**exe** option is followed by the Component Directory in releases **6.0.x**. The **Load** and **Execute** scripts call a Perl script (findexe.pl) to find the corresponding executable.
- The –**exe** option is followed by the fully qualified executable name in releases **6.1** and later.
- **6.1** formalized what comes after the **::Ok::** string. The **Load**, **Unload**, **Execute**, and **Terminate** can succeed (in other words, return **::Ok::**) but may return a warning. The warning is identified by the parameter –**warning** followed by a string enclosed in single quotes ('). The toolset will display a dialog box specifying that a warning occurred. The string returned in quotes is appended to the toolset logs. Anything appearing after the **-passback** parameter will be returned to the originating call.

Third-Party Source Code Debugger Integration

The format for the Debugger Mode is **Debugger-X** where **X** is the name of the debugger DLL. This DLL must exist in the $ROSERT_HOME/bin/$ROSERT_HOST directory and is called **libX.dll**.

Registering Threads on Unix

When building a debugger integration DLL without MainWin and using **callback** functions, additional steps are required to ensure that Rational Rose RealTime knows about the **callback** thread. The following steps are necessary for a thread-safe interface:

- Call **tcThreadInit()** from the **callback** thread before doing any callbacks.
- The **callback** thread must call **tcThreadCleanup()** before terminating.

There is a header file for this service in $ROSERT_HOME/bin/tc/tcsetup.h and a supporting dynamic library (for Solaris) in $ROSERT_HOME/bin/tc/sun5/libtcsetup.so.

You may call **tcThreadInit** (init) and **tcThreadCleanup** (cleanup) as many times as you like, as long as the **tcThreadInit** is always followed by a **tcThreadCleanup** before the next **init** occurs. This is useful if you wanted to do something similar to the following: **tcThreadInit**, **callback**, **tcThreadCleanup**, for each **callback** instead of **tcThreadInit** at thread startup, and **tcThreadCleanup** at thread termination. However, we recommend that the **tcThreadInit** and **tcThreadCleanup** fuctions be called only once (**tcThreadInit** at startup and **cleanup** at termination) since this approach is less error prone.

Calling Sequence

Source code debuggers come with a variety of capabilities. For the toolset to use the debugger DLL in the best possible way, the DLL must provide a list of its capabilities. The following are capabilities of the debugger DLL that are available to Rational Rose RealTime:

Capability	Description
Function Breakpoints	The DLL uses the function name to set a breakpoint.
Line Breakpoints	The DLL uses a file name and line number to set a breakpoint.
Detects Breakpoint Hits	The DLL calls the callback function when a breakpoint is hit.
User Termination Detected	The DLL calls the callback function when it detects that the user terminated the debugger manually.
Debugger Loads Target	The DLL must be called to load the target. If not, the toolset uses the Basic mode mechanism, if one exists.
Debugger Unloads Target	The DLL must be called to unload the target. If not, the toolset uses the Basic mode mechanism, if one exists.
Debugger Executes Component	The DLL must be called to start the Component Instance. If not, the toolset uses the Basic mode mechanism, if one exists.

Capability	Description
Debugger Terminates Component Instance	The DLL must be called to terminate a component instance. If not, the toolset will use the Basic mode mechanism if one exists.
Supports Search Paths	The DLL can use a given search path to search for source code.
Reload Before Restarting	The target must be reloaded before it is restarted.

The values of these flags determine how and which debugger DLL functions are called. The following are the rules of operation.

1. The debugger DLL is loaded once the user applies the change to the Operation Mode in the Component Instance specification for the Component Instance. The debugger DLL is loaded only once per toolset session.
2. If the DLL is loaded successfully, the toolset obtains the debugger DLLs capabilities and saves them.
3. Next, the toolset calls the **tcCreateDebugSession** function to create a new session. **Note:** A new session is created for each Component Instance that uses the debugger DLL.
4. The Target Control capabilities (**Load**, **Unload**, **Run**, **Shutdown**) are determined using the debugger DLL capabilities as well as the Target Control scripts. The debugger DLL capabilities take precedence over the Target Control scripts.
5. If a target must be loaded, it can be loaded in one of two ways: using the debugger or the Basic mode Target Control script. If the "Debugger Loads Target" flag is set, the debugger DLL is expected to load the target in the **tcInitializeDebugger** function. Otherwise, the Target Control load script is used to load the target, and then the **tcInitializeDebugger** function is called.
6. If the target is not loadable, then the **tcInitializeDebugger** function is called when the user invokes the Run command.
7. If the "Debugger Executes Component" flag is set, then the **tcStartDebugger** function is called. If not set, then the Target Control execute script is called and then followed by a call to the **tcStartDebugger** function. **Note**: The breakpoint functions may be called before the **tcStartDebugger** function if breakpoints were set in the previous debug session.

8 When the user invokes the Shutdown command, all breakpoints are removed, and the **tcStopDebugger** function is called. If the "Debugger Terminates Component Instance" is set, the **tcStopDebugger** must terminate the Component Instance. If not set, then the Target Control terminate script is called. If the target does not need to be unloaded, then the **tcCleanupDebugger** function is called as well.

9 When the user invokes the Unload command and the "Debugger Unloads Component" flag is set, then the **tcCleanupDebugger** function is called. This function must unload the component from the target. If this capability is not set, then the Target Control unload script is called.

10 When the Debugger DLL is unloaded from the toolset (when the Component Instance Operation mode is changed or when the toolset is shut down) **tcDestroySession** is called. This function is responsible for releasing any resources associated with this debugger DLL session.

Debugger DLL API

This section describes the API that must be implemented by a debugger DLL. The file, tcdllinterface.h contains all the required type declarations and function prototypes. The functions are:

- Get DLL Capabilities
- Create Debug Session
- Destroy Debug Session
- Initialize Debugger
- Cleanup Debugger
- Start Debugger
- Stop Debugger
- Set Callback
- Event Callback Function
- Set Source Search Path
- Set Breakpoint in File
- Set Breakpoint At Function
- Clear Breakpoint
- Set DllTrace

Note: Several functions have parameters of type **TC_TCHAR**. This type corresponds to **TCHAR** type familiar to Windows developers. It is either a regular character (**char**) or a wide character (**wchar_t**). By default, **TC_TCHAR** is type defined to char in the file tcdllinterface.h.

Get DLL Capabilities

```
TCRET
```

```
tcGetDllCapabilities(
                TCDLLCAPS * pCaps/* Pointer to struct to get the
capabilites */
) ;
```

Description

This function populates in the given capability structure with the capabilities of the corresponding DLL. This is the first function that is called in the debugger DLL.

Arguments

TCDLLCAPS * pCaps	Structure to receive the DLL capabilities.

Returns

TC_OK	Operation was successful.
TC_FAILED	Operation failed. Missing capability structure.

Create Debug Session

```
TCHANDLE
tcCreateDebugSession(
                  const TC_TCHAR * szServerName, /* Name of Target
Server */
                  const TC_TCHAR * szTargetName, /* Name of Target*/
                  const TC_TCHAR * szArchitecture,/* Processor
Architecture */
                  const TC_TCHAR * szOS,          /* Operating System
*/
                  TCDEBUGFLAG      eFlag           /*
Enables/disables Tracing*/
) ;
```

Description

This function is called to create a debug session. It is called after the debugger DLL is loaded. It returns a DLL specific handle that represents the newly created session. This handle is passed back to all other calls except the **tcGetDllCapabilities**. Typically, the handle is a pointer to a DLL-specific structure that maintains session-specific information.

Arguments

const TC_TCHAR * szServerName	Name or address of a Target Server.
const TC_TCHAR * szTargetName	Name or address of the target.
const TC_TCHAR * szArchitecture	Type of CPU on the target.
const TC_TCHAR * szOS	OS running on the target
TCDEBUGFLAG eFlag	Enables/Disables Debug output from the DLL. See Note below.

Returns

TCHANDLE	DLL specific handle identifying the newly created session.
(TCHANDLE)0	Unable to create a session.

Note: Note: Currently, the toolset does not provide any means to set or clear the debug flag.

Destroy Debug Session

```
TCRET
tcDestroyDebugSession(
                TCHANDLE    hSession  /* Session to terminate */
) ;
```

Description

This function is called before the Debugger DLL is unloaded. It must release all session-specific resources that may have been allocated during the session.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session.

Returns

TC_OK	Operation was successful.
TC_FAILED	Operation failed.

Initialize Debugger

```
TCRET
tcInitializeDebugger(
                TCHANDLE       hSession,   /* Debugger Session */
                const TC_TCHAR * szComponent/* Location/name of the
component */
) ;
```

Description

This function is called to identify the component that the debugger is to work with. In some environments this function will load the component onto the target.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session.
const TC_TCHAR * szComponent	The fully qualified name of the component.

Returns

TC_OK	Operation was successful.
TC_FAILED	Operation failed.

Cleanup Debugger

```
TCRET
tcCleanupDebugger(
            TCHANDLE        hSession  /* Debugger Session */
) ;
```

Description

This function is called to undo the activities of the tcInitializeDebugger function. In some environments this function will unload the component from the target.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session.

Returns

TC_OK	Operation was successful.
TC_FAILED	Operation failed.

Start Debugger

```
TCRET
tcStartDebugger(
            TCHANDLE           hSession,/* Debugger Session */
            const TC_TCHAR * pszArgs,/* Command line arguments for
comp */
            int              nPriority   /* start up priority */
) ;
```

Description

This function is called to start the Component Instance. If the debugger does not start the Component instance, this is the point where the debugger should attach to it.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session.
const TC_TCHAR * pszArgs,	Command line arguments for the Component Instance.
int nPriority	Priority to run the application.

Returns

TC_OK	Operation was successful.
TC_FAILED	Operation failed.

Stop Debugger

```
TCRET
tcStopDebugger(
            TCHANDLE        hSession/* Loader.Debugger Session */
) ;
```

Description

This function is called to terminate the Component Instance. If the debugger does not terminate the Component instance, this is the point where the debugger should detach from it.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session.

Returns

TC_OK	Operation was successful.
TC_FAILED	Operation failed.

Set Callback

```
TCRET
tcSetCallback(
            TCHANDLE       hSession,      /* Debugger Session */
            CALLBACKFNC pfncCallback,/* function to call on event */
            USERDEFINED lUserDefined1,/* toolset defined data */
            USERDEFINED lUserDefined2 /* toolset defined data */
) ;
```

Description

This function is called during the Target Observability session if the debugger DLL can detect breakpoint hits or user termination. It is used to set or clear a Toolset defined function.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session.
CALLBACKFNC pfncCallback,	Pointer to function the debugger DLL is to call when a breakpoint hit or user termination is detected.
USERDEFINED lUserDefined1	Toolset information that must be passed back in the callback function.
USERDEFINED lUserDefined2	Toolset information that must be passed back in the callback function.

Returns

TC_OK	Operation was successful.
TC_FAILED	Operation failed.

Event Callback Function

```
void
fncCallback(
                    TCDLLEVENT* pEvent, /* identifies what event
occurred */
                    USERDEFINED  data1, /* data from SetCallback */
                    USERDEFINED  data2 /* data from SetCallback */
) ;
```

Description

This is the prototype of the callback function that is to be called by the debugger DLL when a breakpoint hit or user termination is detected.

Arguments

TCDLLEVENT * pEvent	Identifies the type of event the Debugger DLL is notifying the toolset of.
USERDEFINED lUserDefined1	Toolset information from the last tcSetCallback.
USERDEFINED lUserDefined2	Toolset information from the last tcSetCallback.

Returns

void	Nothing

Set Source Search Path

```
TCRET
tcSetSearchPath(
            TCHANDLE            hSession, /* Debugger Session */
            int               nEntries, /* number of paths */
           const TC_TCHAR ** ppszSearchPaths/* list of search paths */
) ;
```

Description

This function is called by the toolset to specify the directories that contain the generated source code.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session.
int nEntries	The number of paths specified in the next parameter.
const TC_TCHAR ** ppszSearchPaths	A list of search paths.

Returns

TC_OK	Operation was successful.

TC_FAILED	Operation failed.

Set Breakpoint in File

```
unsigned long
tcSetBreakpointInFile(
            TCHANDLE           hSession, /* Debugger Session */
            const TC_TCHAR * szFileName,/* File to set breakpoint in
*/
            int              nLineNo  /* line number in file */
) ;
```

Description

This function is called when a breakpoint is required and the Debugger DLL supports breakpoints using file name and line number. This function may be called before the tcStartDebugger.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session.
const TC_TCHAR * szFileName	Name of file where we want to set the breakpoint.
int nLine	No The line number in the file where the breakpoint is to be set.

Returns

unsigned long	A number uniquely identifying the corresponding breakpoint.
0	Unable to set breakpoint.

Set Breakpoint At Function

```
unsigned long
tcSetBreakpointAtFnc(
            TCHANDLE             hSession,      /* Debugger Session */
            const TC_TCHAR    * szFunctionName/* fully qualified name
*/
```

```
) ;
```

Description

This function is called when a breakpoint is required and the Debugger DLL supports breakpoints using function names. This function may be called before the **tcStartDebugger.**

Arguments

TCHANDLE hSession	A handle identifying a particular debug session.
const TC_TCHAR * szFunctionName	The fully qualified name of the function.

Returns

unsigned long	A number uniquely identifying the corresponding breakpoint.
0	Unable to set breakpoint.

Clear Breakpoint

```
TCRET
tcClearBreakpoint(
            TCHANDLE           hSession,       /* Debugger Session */
            unsigned long    nBreakpointId   /* breakpoint to remove
*/
) ;
```

Description

This function is removes the specified breakpoint for the given session.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session.
unsigned long nBreakpointId	Identifier of breakpoint to be removed. Returned by a set breakpoint function.

Returns

TC_OK	Operation was successful.
TC_FAILED	Unable to remove breakpoint.

Set DllTrace

```
void
tcSetDllTrace(
            TCHANDLE        hSession,/* Debugger Session */
            TCDEBUGFLAG     eFlag   /* enables/disables trace output
*/
) ;
```

Description

This function is enables or disables Debugger DLL output for the given session.

Arguments

TCHANDLE hSession	A handle identifying a particular debug session.
TCDEBUGFLAG eFlag	Specifies whether to enable or disable output.

Returns

TC_OK	Operation was successful.
TC_FAILED	Operation failed.

Note: This function is not currently used by the toolset, but it must exist. If this function is omitted from the debugger DLL, the toolset will not load the DLL successfully.

Index

O

R

S

T

U